УДК 004.4
ББК 32.973
О 75

Автор-составитель С. В. Кравченко, канд. физ.-мат. наук, доцент

Рецензенты: Е. В. Коробейникова, канд. физ.-мат. наук, начальник сектора внедрения и поддерки SAP РУП ПО «Белоруснефть»;
О. И. Еськова, канд. техн. наук, доцент Белорусского торгово-экономического университета потребительской кооперации

Рекомендовано к изданию научно-методическим советом учреждения образования «Белорусский торгово-экономический университет потребительской кооперации». Протокол № 4 от 11 апреля 2017 г.

О 75 **Основы** объектно-ориентированного программирования : пособие для реализации содержания образовательных программ высшего образования I ступени / авт.-сост. С. В. Кравченко. – Гомель : учреждение образования «Белорусский торгово-экономический университет потребительской кооперации», 2018. – 208 с.
ISBN 978-985-540-438-6

В издании излагаются базовые теоретические сведения о программировании на языке Java. Приведены задания для выполнения.

Пособие предназначено для студентов специальности 1-28 01 01 «Экономика электронного бизнеса».

УДК 004.4
ББК 32.973

ISBN 978-985-540-438-6

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Цель учебной дисциплины заключается в получении студентами базовых научно-технических знаний и приобретении начальных практических навыков разработки объектно-ориентированных программ.

В результате изучения учебной дисциплины студент должен знать следующее:

- базовые понятия и синтаксис языка, технологию объектно-ориентированного программирования;
- методы определения и использования основных объектов и конструкций языка;
- основные технологические приемы разработки программ;
- технологию организации и использования иерархии классов, предопределенных классов и типов данных;
- оптимальные методы использования объектно-ориентированного языка для решения прикладных задач по направлениям.

В результате изучения учебной дисциплины студент должен:

- использовать технологию объектно-ориентированного программирования;
- определять программные абстракции, модули, строить иерархию классов для реализации программ;
- создавать свои и использовать предоставляемые стандартные библиотеки шаблонов сложных структур данных;
- применять методы инкапсуляции, наследования, полиморфизма и другие сложные абстракции объектно-ориентированного программирования для разработки сложных программ и систем по направлениям;
- владеть методами объектно-ориентированного программирования.

Тема 1. ЭЛЕМЕНТАРНЫЕ ТИПЫ ДАННЫХ JAVA

Основные теоретические сведения

Типы данных в Java подразделяются на простые (элементарные, примитивные) и ссылочные. К простым типам данных относятся типы для работы с целыми числами (типы byte, short, int, long), действительными числами, числами с плавающей точкой (типы float, double), символьный (char) и логический (boolean) типы. К ссылочным типам данных относятся массивы, классы и интерфейсы.

Диапазон значений, количества байт, выделяемых в памяти для записи значения соответствующего типа, и значения по умолчанию приведены в таблице 1.

Таблица 1 – **Элементарные типы данных**

Тип	Размер, байт	Диапазон значений	Значение по умолчанию	Описание
byte	1	–128 … 127	0	Самое маленькое целое
short	2	–32 768 … 32 767	0	Короткое целое
int	4	–2 · 109 … 2 · 109	0	Целое число
long	8	–9 · 1 018 … 9 · 1 018	0L	Длинное целое
float	4	–10 127 … 10 127	0.0f	Дробное
double	8	–101 023 … 101 023	0.0d	Дробное двойной длины
boolean	1	true, false	false	Логический тип
char	2	0 … 65 535	'\u0000'	Символы

При объявлении переменной перед ее именем обязательно указывается идентификатор следующего типа:

```
тип имя_переменной;
```

Например, конструкцией `int var1;` объявлена переменная *var1* целочисленного типа *int.*

После объявления переменная может быть инициализирована, ей может быть присвоено значение. В качестве оператора присваивания в Java используется оператор =. Можно привести следующий пример:

```
int var1;
var1 = 23;
```

Инициализация переменных может выполняться сразу при их объявлении, например,

```
int count = 10;
char ch = ‘S’;
float f = 1.2F;
int a, b = 8, c = 19, d;
```

Все операторы Java можно разделить на следующие группы: арифметические, логические, побитовые и сравнения (таблицы 2–4).

Таблица 2 – **Арифметические операторы Java**

Оператор	Пример	Пояснение
+	a + b	Оператор сложения
–	c – d	Оператор вычитания
*	a * b	Оператор умножения
/	a / b	Оператор деления
%	c % d	Остаток от деления первого числа на второе
–	–a	Одинарный минус. Меняет знак числа на противоположный
+	+a	Одинарный плюс
+=	a += b;	Сложение (упрощенная форма с присваиванием). Эквивалентно команде a = a + b;
–=	a –= b;	Вычитание (упрощенная форма с присваиванием). Эквивалентно команде a = a – b;
*=	a *= b;	Умножение (упрощенная форма с присваиванием). Эквивалентно команде a = a * b;
/=	a /= b;	Деление (упрощенная форма с присваиванием). Эквивалентно команде a = a / b;
%=	a %= b;	Остаток от деления (упрощенная форма с присваиванием). Эквивалентно команде a = a % b;
++	a++ ++a	Инкремент. Увеличивает значение переменной на 1
– –	a– – – –a	Декремент. Уменьшает значение переменной на 1

Таблица 3 – **Логические операторы Java**

Оператор	Пример	Пояснение
&	a & b	Логическое И
&&	c && d	Укороченное И. Если значение первого операнда равно *false*, то значение второго не проверяется, результат операции получает значение *false*
|	a | b	Логическое ИЛИ
||	a || b	Укороченное ИЛИ. Если значение первого операнда равно *true*, то значение второго не проверяется и результат операции получает значение *true*
^	c ^ d	Исключающее ИЛИ. Результатом операции является *true*, только если значение одного операнда равно *true*. Иначе результат операции получает значение *false*
!	! a	Логическое отрицание

Таблица 4 – **Операторы сравнения Java**

Оператор	Пример	Пояснение
<	a < b	Проверяет, что a меньше b
<=	c <= d	Проверяет, что c меньше или равно d
>	a > b	Проверяет, что a больше b
>=	c >= d	Проверяет, что c больше или равно d
= =	i = = j	Проверяет, что i равно j
!=	a != b	Проверяет, что операнды имеют разные значения

Если в выражение входят переменные разных типов, то вначале нужно выполнить преобразование переменных к общему формату. Преобразование типов осуществляется автоматически, если типы совместимы и целевой тип больше по диапазону, чем исходный. Явное приведение типов – это команда компилятору преобразовать результат вычисления выражения в указанный тип. Общий синтаксис явного приведения типов имеет следующий вид:

```
(целевой_тип) выражение.
```

Можно привести следующий пример преобразования типов:

```
int i = 260;
byte и = (byte) i; // в этом случае b примет значение 4.
```

Пример решения задачи

Программа рассчитывает расстояние в метрах до места удара молнии. Известно количество секунд, прошедших между вспышкой молнии и ударом грома. Текст программы имеет следующий вид:

```
class Sound {
   public static void main(String args[ ]) {
      int t = 13;
      int s = 330 * t;
      System.out.println("Расстояние до места вспышки мол-
      нии " +
         "составляет " + s + " метров");
   }
}
```

Задания для самостоятельной работы

Задания выполняются в папке (package) с именем *theme1*.

Задание 1.1. Используя текст программы из примера напишите программу, которая бы вычисляла расстояние до крупного объекта, например, скалы, по времени прихода эхо.

Задание 1.2. Наберите нижеприведенный текст программы. Укажите, какой результат даст выполнение этой программы.

```
// Демонстрация приведения типов
class CastDemo {
   public static void main(String args[ ]) {
      double x = 10.0, y = 3.0;
      byte b;
      int i;
      char ch;

      //приведение типа double к типу int
      //теряется дробная часть числа
      i = (int)(x / y);
      System.out.println("i = " + i);
      //приведение типа byte к типу int
      //тип byte может содержать значение 100
```

```
      i = 100;
      b = (byte) i;
      System.out.println("b = " + b);
      //приведение типа byte к типу int
      //тип byte может не содержать значение 257
      i = 257;
      b = (byte) i;
      System.out.println("b = " + b);
      //представление символа X в коде ASCII
      b = 88;
      ch = (char) b;
      System.out.println("ch: " + ch);
   }
}
```

Задание 1.3. В нижеприведенном тексте программы расставьте правильно операторы приведения типа, чтобы переменная d была положительной:

```
public class Exercice1_3 {
   public static void main(String[ ] args) {
      int a = 0;
      int b = (byte) a + 46;
      byte c = (byte) (a*b);
      double f = (char) 1234.15;
      long d = (short) (a + f / c + b);
      System.out.println(d);
   }
}
```

Задание 1.4. В нижеприведенном тексте программы расставьте правильно операторы приведения типа, чтобы было выведено на консоль 3.765:

```
public class Exercice1_4 {
   public static void main(String[ ] args) {
      int a = 15;
      int b = 4;
      float c = a / b;
      double d = a * 1e-3 + c;
```

```
      System.out.println(d);
   }
}
```

Задание 1.5. В нижеприведенном тексте программы добавьте одну операцию по преобразованию типа, чтобы переменная b была равна 0:

```
public class Exercice1_5 {
   public static void main(String[ ] args) {
      float f = (float)128.50;
      int i = (int)f;
      int b = (int)(i + f);
      System.out.println(b);
   }
}
```

Задание 1.6. В нижеприведенном тексте программы добавьте одну операцию по преобразованию типа, чтобы было выведено на консоль 9:

```
public class Exercice1_6 {
   public static void main(String[ ] args) {
      short number = 9;
      char zero = '0';
      int nine = (zero + number);
      System.out.println(nine);
   }
}
```

Задание 1.7. В нижеприведенном тексте программы расставьте правильно операторы приведения типа, чтобы было на консоль выведено 256:

```
public class Exercice1_7 {
   public static void main(String[ ] args) {
      int a = (byte)44;
      int b = (byte)300;
      short c = (byte)(b - a);
      System.out.println(c);
   }
}
```

Задание 1.8. В нижеприведенном тексте программы добавьте одну операцию по преобразованию типа, чтобы было выведено на консоль 2.941:

```
public class Exercice1_8 {
   public static void main(String[ ] args) {
      int a = 50;
      int b = 17;
      double d = a / b;
      System.out.println(d);
   }
}
```

Задание 1.9. В нижеприведенном тексте программы добавьте одну операцию по преобразованию типа, чтобы было выведено на консоль 5.5:

```
public class Exercice1_9 {
   public static void main(String[ ] args) {
      int a = 5;
      int b = 4;
      int c = 3;
      int e = 2;
      double d = a + b/c/e;
      System.out.println(d);
   }
}
```

Задание 1.10. В нижеприведенном тексте программы добавьте одну операцию по преобразованию типа, чтобы было выведено на консоль 1.0:

```
public class Exercice1_9 {
   public static void main(String[ ] args) {
      int a = 257;
      int b = 4;
      int c = 3;
      int e = 2;
      double d = a + b/c/e;
      System.out.println(d);
   }
}
```

Тема 2. УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ JAVA

Основные теоретические сведения

Полная форма условного оператора (оператора ветвления) имеет следующий вид:

```
if (условие) оператор;
else оператор;
```

Оператор после *else* не является обязательным. Также существует возможность использовать вложенные операторы *if*. Использование условного оператора можно представить следующим образом:

```
int num1 = 6;
int num2 = 8;
if (num1 < num2)
   System.out.println("Первое число больше второго");
else if (num1 > num2)
   System.out.println("Второе число больше первого");
else System.out.println("Числа равны");
```

Вместо оператора *if* может использоваться тернарный оператор (оператор ?). Его синтаксис имеет следующий вид:

```
a ? b : c;
```

Эта запись эквивалентна следующей:

```
if (a)
b
else c
```

Например, модуль числа с помощью тернарного оператора может быть вычислен следующим образом:

```
absValue = value >- 0 ? value : -value ;
```

Если требуется проверить множество альтернатив, то вместо нескольких вложенных операторов *if* используют оператор *switch*. Синтаксис оператора *switch* следующий:

```
switch (условие) {
case константа1 :
последовательность операторов
break;
case константа2 :
последовательность операторов
break;
…
default :
последовательность операторов
}
```

Выражение, управляющее оператором *switch*, должно быть типом *byte*, *short*, *int*, *char*, перечислением или символьной строкой.

Последовательность операторов из ветви *default* выполняется в том случае, если ни одна из констант выбора не совпадает с заданным выражением. Ветвь *default* не является обязательной.

При выполнении оператора *break* оператор *switch* завершает работу, управление передается следующему за ним оператору. Если в некоторой ветви отсутствует оператор *break*, то сначала выполняются все операторы в этой ветви, затем выполняются операторы, совпадающие с константой выбора в следующей ветви *case*. Этот процесс продолжается до тех пор, пока не встретится оператор *break* или не будет достигнут конец оператора *switch*.

Использования оператора *switch* можно представить следующим образом:

```
int num = 8;
switch(num) {
   case 1 :
      System.out.println("Число равно 1");
      break;
   case 8 :
      System.out.println("Число равно 8");
      break;
   case 9 :
      System.out.println("Число равно 9");
      break;
   default :
      System.out.println("Число не равно 1, 8 или 9");
}
```

Для выполнения однотипных многократно повторяющихся действий используют операторы цикла. В Java существует несколько операторов цикла. Синтаксис оператора *for* имеет следующий вид:

```
for (инициализация_счетчика; условие; изменение_счетчика) {
   // тело цикла
}
```

Синтаксис оператора *while* следующий:

```
while (условие) {
   // тело цикла
}
```

В качестве примера использования цикла *while* можно рассмотреть следующий пример вывода букв английского алфавита:

```
char ch = ‘a’;
while (ch <= ‘z’) {
   System.out.println(ch);
   ch++ ;
}
```

Синтаксис оператора *do-while* имеет следующий вид:

```
do {
   // тело цикла
} while (условие) ;
```

Если число итераций заранее известно, то лучше выбрать цикл *for*. Цикл *while* оказывается наиболее удобным тогда, когда число повторений цикла заранее неизвестно. В случаях, когда требуется, чтобы была выполнена хотя бы одна итерация, используют цикл *do-while*.

Для завершения цикла можно использовать команду *break*. Когда в теле цикла встречается оператор *break*, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла. Оператор *break* можно применять в любых циклах.

Оператор *continue* используется для завершения текущей итерации цикла и перехода к следующей.

Задания для самостоятельной работы

Задания выполняются в папке (package) с именем *theme2*.

Задание 2.1. На рисунке 1 даны части одной программы. Соберите их в правильном порядке, добавляя недостающие фигурные скобки, чтобы в результате работы программы на консоль было выведено следующее:

```
a-b c-d
```

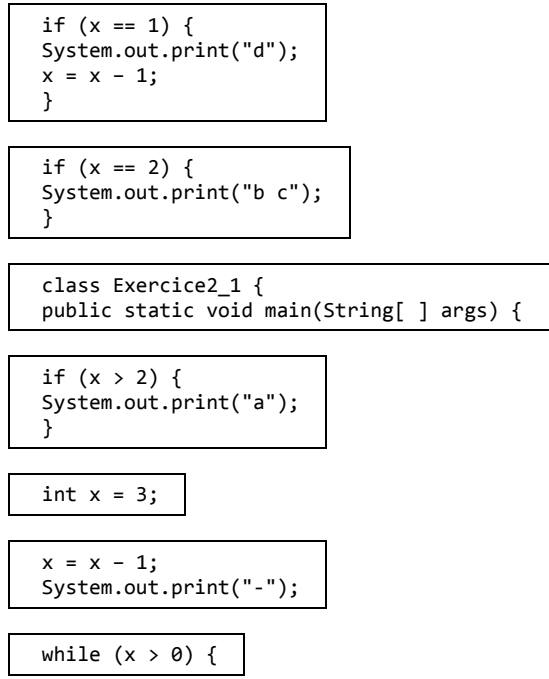

Рисунок 1 – **Части программы к заданию 2.1**

Задание 2.2. Ниже представлены три программы. Исправьте в них ошибки, чтобы программы компилировались.

Программа А

```
class Exercise1A {
```

```
   public static void main(String[ ] args) {
      int x = 1;
      while (x < 10) {
         if ( x > 3) {
            System.out.println("большой x");
         }
      }
   }
}
```

Программа B

```
public static void main(String[ ] args) {
   int x = 5;
   while (x > 1) {
      x = x – 1;
      if ( x < 3) {
         System.out.println("маленький икс");
      }
   }
}
```

Программа C

```
class Exercise1C {
   int x = 5;
   while (x > 1) {
      x = x – 1;
      if ( x < 3) {
         System.out.println("маленький икс");
      }
   }
}
```

Задание 2.3. На рисунке 2 даны части одной программы. Соберите их в правильном порядке, добавьте недостающие фигурные скобки, чтобы в результате работы программы на консоль было выведено следующее:

0 4
0 3

1 4
1 3
3 4
3 3

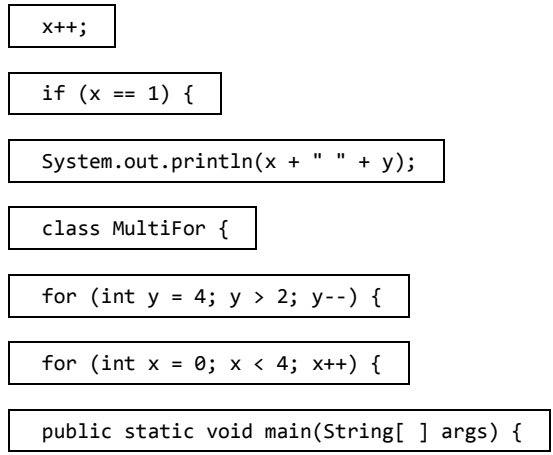

Рисунок 2 – **Части программы к заданию 2.3**

Задание 2.4. Используя цикл *for* выведите на экран прямоугольный треугольник из восьмерок со сторонами 10 и 10. Пример вывода на консоль следующий:

8
88
888
...
8888888888

Тема 3. ВВЕДЕНИЕ В КЛАССЫ, ОБЪЕКТЫ, МЕТОДЫ

Основные теоретические сведения

Класс представляет собой шаблон, по которому определяется вид объекта. В нем указываются данные и код, который будет оперировать этими данными. Объекты – это экземпляры класса. Класс фактически представляет собой описание, в соответствии с которым должны создаваться объекты. Класс – это логическая абстракция. Физиче-

ское представление класса в оперативной памяти появляется лишь после того, как будет создан объект этого класса.

Класс создается с помощью ключевого слова *class*. Общая форма определения класса имеет следующий вид:

```
class имя_класса {
   // объявление переменных экземпляра
   тип переменная1;
   тип переменная2;
   // …
   тип переменнаяN;

   // объявление методов
   тип метод1 (параметры) {
      // тело метода1
   }
   тип метод2 (параметры) {
      // тело метода2
   }
   // …
   тип методN (параметры) {
      // тело методаN
   }
}
```

При создании объекта используется следующий синтаксис:

```
имя_класса имя_объекта ;
имя_объекта = new имя_класса( ) ;
```

Обычно эти две команды объединяют следующим образом:

```
имя_класса имя_объекта = new имя_класса( ) ;
```

Каждый раз, когда создается экземпляр класса, создается объект, содержащий собственные копии всех переменных экземпляра (т. е. нестатических переменных), определенных в классе.

Для обращения к членам класса используется точечная нотация (оператор точка), которая имеет следующий вид:

```
объект.член
```

Метод состоит из одной или нескольких команд. У каждого метода есть свое имя, которое используется для его вызова. Общий синтаксис объявления метода имеет следующий вид:

```
возвращаемый_тип имя (список_параметров) {
//тело метода
}
```

Здесь *возвращаемый_тип* обозначает тип данных, которые возвращает метод. Если метод ничего не возвращает, то для него указывается тип *void*. В качестве имени метода может использоваться любой допустимый идентификатор, не приводящий к конфликту имен в текущей области действия. В скобках указывается *список_параметров* – это последовательность параметров, разделенных запятыми, для каждого из которых указывается тип и имя.

Для выполнения явного возврата из метода используется оператор *return*. Он снова передает управление объекту, который вызвал данный метод. Этот оператор относится к операторам перехода. Если в сигнатуре метода тип возвращаемого значения не *void*, то в теле метода должен быть хотя бы один оператор

```
return выражение;
```

В операторе тип выражения должен совпадать с типом возвращаемого значения. Этот оператор возвращает результат вычисления выражения в точку вызова метода.

Если тип возвращаемого значения – это *void*, то возврат из метода выполняется либо после выполнения последнего оператора тела метода, либо в результате выполнения оператора

```
return;
```

Таких операторов в теле метода может быть несколько.

Конструктор инициализирует объект при его создании. Имя конструктора совпадает с именем класса, с точки зрения синтаксиса он подобен методу. У конструктора нет возвращаемого типа, указываемого явно. Параметры в конструктор вводятся также, как и в метод.

Задания для самостоятельной работы

Задания выполняются в папке (package) с именем *theme3*.

Задание 3.1. Определите, что выведет на экран следующая программа:

```
class Output {
   public static void main (String[ ] args0 {
      Output o = new Output( );
      o.go( );
   }

   void go( ) {
      int y = 7;
      for (int x = 1; x < 8; x++) {
         y++;
         if (x > 4) {
            System.out.print(++y + " ");
         }
         if (y > 14) {
            System.out.println(" x = " + x);
            break;
         }
      }
   }
}
```

Задание 3.2. В папке *theme3* создайте папку (package) с именем *exercise3_2*.

В папке *exercise3_2* создайте класс *Vehicle* (транспортное средство).

В классе *Vehicle* создайте переменные *model* (модель) типа *String*, *passengers* (количество пассажиров) типа *int*, *fuelTank* (емкость топливного бака) типа *int*, *lpkm* (потребление топлива в литрах на 100 км) типа *double*.

В классе *Vehicle* создайте параметризованный конструктор, позволяющий инициализировать все четыре переменные.

В классе *Vehicle* создайте метод *range*( ) без параметров, позволяющий рассчитать дальность поездки транспортного средства с пол-

ным топливным баком. Определите, какой должен быть возвращаемый тип у этого метода.

В классе *Vehicle* создайте метод *fuelNeeded*( ). Он получает в качестве параметра расстояние в километрах, которое должно проехать транспортное средство, а возвращает необходимое для этого количество топлива в литрах.

В папке *exercise3_2* создайте класс *VehicleTest*.

В классе *VehicleTest* создайте метод *main*.

В методе *main* создайте три экземпляра класса (объекта) *Vehicle*.

Для каждого экземпляра класса выведите сведения о нем на основании нижеприведенного примера:

LADA XRAY может перевезти 5 пассажиров на расстояние 694,44 км.

LADA XRAY: для преодоления 300 км потребуется 21,6 л топлива.

Задание 3.3. В папке *theme3* создайте две папки (package) с именами *exercise3_3A* и *exercise3_3B*. В папке *exercise3_3A* создайте два класса *TapeDeck* и *TapeDeckTestDrive*, в папке *exercise3_3B* создайте два класса *DVDPlayer* и *DVDPlayerTestDrive*. Определите, скомпилируются ли приведенные ниже программы. Устраните имеющиеся ошибки и посмотрите на результат работы следующих программ:

Программа А

```
class TapeDeck {

   boolean canRecord = false;

   void playTape( ) {
      System.out.println("пленка проигрывается");
   }

   void recordTape( ) {
      System.out.println("идет запись на пленку");
   }
}

class TapeDeckTestDrive {
   public static void main(String[ ] args) {
      t.canRecord = true;
```

```
      t.playTape( );
      if (t.canRecord = = true) {
         t.recordTape( );
      }
   }
}
```

Программа В

```
class DVDPlayer {

   boolean canRecord = false;

   void recordDVD( ) {
      System.out.println("идет запись DVD");
   }
}

class DVDPlayerTestDrive {
   public static void main(String[ ] args) {
      DVDPlayer d = new DVDPlayer( );
      d.canRecord = true;
      d.playDVD( );

      if (t.canRecord = = true) {
         t.recordDVD( );
      }
   }
}
```

Задание 3.4. В папке *theme3* создайте папку (package) с именем *exercise3_4*, в которой выполните следующее задание. На рисунке 3 приведены отдельные части кода. Соберите его в правильном порядке, чтобы вывод на консоль имел следующий вид:

бах бах ба-бах
динь динь ди-динь

```
d.playSnare( );
```

```
DrumKit d = new DrumKit( );
```

```
boolean topHat = true;
boolean snare = true;
```

```
void playSnare( ) {
System.out.println("бах бах ба-бах");
}
```

```
public static void main(String[ ] args) {
```

```
if (d.snare == true) {
d.playSnare( );
}
```

```
d.snare = false;
```

```
class DrumKitTestDrive {
```

```
d.playTopHat( );
```

```
class DrumKit {
```

```
void playTopHat( ) {
System.out.println("динь динь ди-динь ");
}
```

Рисунок 3 – **Части программы к заданию 3.4**

Задание 3.5. В папке *theme3* создайте две папки (package) с именами *exercise3_5A* и *exercise3_5B*. В папке *exercise3_5A* создайте класс *XCopy*, в папке *exercise3_5B* создайте классы *Clock* и *ClockTestDrive*. Определите, скомпилируются ли программы. Устраните имеющиеся ошибки и посмотрите на результат работы программ.

Программа А

```
class XCopy {
   public static void main(String[ ] args) {
```

```
      int orig = 42;
      XCopy x = new XCopy( );
      int y = x.go(orig);
      System.out.println(orig + " " + y);
   }

   int go(int arg) {
      arg = arg * 2;
      return arg;
   }
}
```

Программа B

```
class Clock {
   String time;

   void setTime(String t) {
      time = t;
   }

   void getTime( ) {
      return time;
   }
}

class ClockTestDrive {
   public static void main(String[ ] args) {
      Clock c = new Clock( );

      c.setTime("12345");
      String tod = c.getTime( );
      System.out.println("время: " + tod);
   }
}
```

Задание 3.6. В папке *theme3* создайте папку (package) с именем *exercise3_6*.

В папке *exercise3_6* создайте класс *Human* (человек).

В классе *Human* создайте переменные *name* (имя) типа *String*, *sex* (пол) типа *boolean*, *age* (возраст) типа *int*.

В классе *Human* создайте параметризованный конструктор, позволяющий инициализировать все три переменные.

В классе *Human* создайте метод *hello*( ), который выводит на консоль фразу «Привет! Меня зовут <имя>». Укажите, какой должен быть возвращаемый тип у этого метода.

В папке *exercise3_6* создайте класс *HumanTest*.

В классе *HumanTest* создайте метод *main*.

В методе *main* создайте три экземпляра класса (объекта) *Human*.

Для каждого экземпляра класса выведите сведения о нем аналогичного следующему примеру:

Привет! Меня зовут Вася, возраст – 19, пол – мужской.

Тема 4. КЛАСС STRING

Основные теоретические сведения

В Java строки – это объекты. Объект типа *String* определяет символьную строку и поддерживает операции над ней. Создать объект типа *String* можно следующими способами: также, как и объекты других типов с использованием оператора *new* либо с помощью инициализации последовательностью следующих символов:

```
String str = new String("Привет!");
String s = "Привет!";
```

Операция конкатенации (операция объединения строк) обозначается знаком «+». В качестве примера можно привести следующее:

```
String s1 = "Сту";
String s2 = "дент";
String s3 = s1 + s2;
```

Некоторые основные методы класса *String* приведены в таблице 5.

Таблица 5 – **Методы для работы со строками**

Метод	Описание	Пример	Результат
int length()	Длина строки	String str = "университет"; int n1 = srt.	11

Продолжение таблицы 5

Метод	Описание	Пример	Результат
char chatAt(int index)	Возвращает символ на указанной позиции в строке	char c = str.charAt(1);	н
boolean equals (Object o)	Сравнение строк	boolean b1 = str.equals("Университет");	false
boolean equalsIgnoreCase (Object o)	Сравнение строк без учета регистра букв	String s1 = "Университет"; boolean b2 = str. equalsIgnore-Case(s1);	true
String toUpperCase()		String s2 = srt. toUp-perCase();	УНИВЕР-СИТЕТ
String toLowerCase()		String s3 = s1.toLowerCase();	университет
String trim()	Удаляет пробелы в начале и конце строки	String s4 = " лекция "; String s5 = s3.trim();	лекция
String substring(int from, int to)	Возвращает подстроку, заданную начальным и конечным номерами символов; последний символ при этом не входит в подстроку	String s6 = str. substring(1, 3);	ни
String substring(int from)	Возвращает подстроку от переданного номера и до конца строки	String s7 = str. substring(8);	тет
int indexOf(String s)	Поиск подстроки	int n2 = str.indexOf("си");	6
int indexOf(String s, int n)	Поиск подстроки с указанного номера	int n3 = str.indexOf("си", 2);	6
int lastIndexOf (String s)	Поиск подстроки с конца строки	int n4 = str.lastIndexOf("и");	7
int lastIndexOf(String s, int n)	Поиск подстроки с конца строки и до указанного номера	int n4 = str.lastIndexOf("и", 4);	7
String replace(char oldChar, char new-Char)	Замена одного символа на другой	String s8 = str.replace(‘у', ‘У’);	Университет

Окончание таблицы 5

Метод	Описание	Пример	Результат
String replaceAll(String s, String replacement)	Замена одной подстроки на другую		

Пример, демонстрирующий ввод строки с клавиатуры, имеет следующий вид:

```
   public static void main (String[ ] args) throws IOExcep-
tion {
   BufferedReader b =
   new BufferedReader(new InputStreamReader(System.in));
   String s = b.readLine();
   b.close( );
   }
```

Если с клавиатуры вводится целое число, то в пример выше нужно добавить следующую команду:

```
   int n = Interger.parseInt(s);
```

Задания для самостоятельной работы

Задание 4.1. В нижеприведенном тексте программы в строках, в которых задается команда вывода на консоль, в комментариях укажите результат вывода.

```
   public class ExampleStr {
      public static void main(String[ ] args) {
         //кодировка должна быть UTF-8
         String s1 = "кот";
         String s2 = "котенок";

         //длина строки
         int n1 = s1.length(); //
         int n2 = s2.length(); //
         System.out.println(n1);
         System.out.println(n2);
```

```
        //конкатенация строк
        String str1 = "Васька";
        String str2 = s1 + " " + str1;
        System.out.println(str2); //

        //Java сама может привести объекты к строковому
        // представлению, даже если они не являются строками
        int digit = 1;
        String str3 = digit + " " + s1; // число 1 преобра-
зовано в строку "1"
        System.out.println(str3); //

        //отобразить s2 посимвольно через пробел
        for (int i = 0; i < s2.length(); i++) {
           char c = s2.charAt(i);
           System.out.print(c + " ");
        }
        System.out.println();

        //поиск подстроки
        String str4 = "скотина";
        int n3 = str4.indexOf(s1);
        System.out.println(n3); //
        int n4 = s2.indexOf('o');
        System.out.println(n4); //
        int n5 = s2.lastIndexOf('o');
        System.out.println(n5); //

        boolean b1 = str4.contains(s1);
        System.out.println(b1); //

        //выделение подстроки
        String str5 = str4.substring(1, 4); //
        System.out.println(str5); //
        System.out.println(str4.substring(0, 4)); //
        System.out.println(str4.substring(3)); //
        System.out.println(str4); //

        //сравнение строк
        boolean b2 = s1.equals(str5);
        System.out.println(b2); //
```

```
        boolean b3 = s1.equalsIgnoreCase("Кот");
        System.out.println(b3); //

        //метод compareTo
        if (s1.compareTo(s2) == 0)
           System.out.println("s1 и s2 равны");
        else if (s1.compareTo(s2) < 0)
             System.out.println("s1 меньше s2");
             else if (s1.compareTo(s2) > 0)
                System.out.println("s1 больше s2");

        //замена одной подстроки на другую
        String str6 = s1.replace("о", "и");
        System.out.println(str6); //
    }
}
```

Задание 4.2. Напишите программу-игру для угадывания чисел. В папке *theme3* создайте папку (package) с именем *exercise3_6*. В папке *exercise3_6* создайте класс *Player* (игрок). В классе *Player* создайте переменную *number* типа *int* и присвойте ей начальное значение, равное нулю. В этой переменной будет храниться вариант числа, названного игроком.

В классе *Player* создайте метод *guess*( ) (угадать) типа *void*. В методе переменной *number* присваивается случайным образом сгенерированное число от 0 до 9, а затем это число выводится на консоль после фразы «Я думаю, это число».

В папке *exercise3_6* создайте класс *GuessGame*. В классе *GuessGame* создайте три переменных типа *Player*.

В классе *GuessGame* создайте метод *startGame*( ) типа *void*. В методе *startGame*( ) инициализируйте переменные тремя разными объектами класса *Player*.

Объявите три переменные типа *int* для хранения вариантов от каждого игрока. Присвойте им первоначальное значение, равное нулю.

Объявите три переменные типа *boolean* для хранения правильности или неправильности (*true* или *false*) ответов игроков. Укажите, какими должны быть первоначальные значения этих переменных.

Выведите на консоль фразу «Я загадываю число от 0 до 9».

Считайте с консоли число, которое игроки должны угадать, присвойте его переменной *targetNumber*.

Организуйте цикл, пока не будет угадано введенное пользователем число.

Выведите на консоль фразу «Число, которое нужно угадать – <*targetNumber*>».

Вызовите метод *guess*( ) у каждого из объектов класса *Player*.

Переменным, в которых хранятся варианты от каждого игрока, присвойте соответствующие значения. Таким образом, извлекаются варианты каждого игрока (результаты работы их методов *guess*( )), получая доступ к их переменным *number*.

Выведите на консоль фразу «Первый игрок думает, что это …».

Выведите на консоль фразу «Второй игрок думает, что это …».

Выведите на консоль фразу «Третий игрок думает, что это …».

Проверьте вариант каждого игрока на соответствие загаданному числу. Если игрок угадал, то необходимо присвоить соответствующей переменной значение *true*.

Если первый, второй или третий игрок угадал, то выведите на консоль фразу «У нас есть победитель!», затем выведите на консоль номер угадавшего игрока, потом выведите на консоль фразу «Конец игры!».

Если никто не угадал, то нужно продолжать игру, просить игроков сделать еще одну попытку. Выведите на консоль фразу «Игроки должны попробовать еще раз».

В папке *exercise3_6* создайте класс *GameLauncher*.

В классе *GameLauncher* создайте метод *main*.

В методе *main* объявите переменную *game* типа *GuessGame* и инициализируйте ее объектом класса *GuessGame*.

У объекта *game* вызовите метод *startGame*( ).

Тема 5. МАССИВЫ

Основные теоретические сведения

Массив представляет собой совокупность однотипных переменных с общим именем. В Java массивы могут быть одномерными и многомерными. Главное преимущество массивов – возможность организации данных таким образом, чтобы ими было проще манипулировать. У массивов в Java есть одна особенность: они реализованы в виде объектов.

Массив представляет собой объект, в котором имя массива – объектная ссылка. Элементами массива могут быть значения базового типа или объекты. Индексирование начинается с нуля.

Одномерный массив – это такой массив, в котором идентификация элементов осуществляется с помощью одного индекса. Одномерный массив можно объявлять одним из следующих способов:

```
тип[ ] имя_массива ;

тип имя_массива [ ] ;
```

Этот оператор лишь объявляет переменную, а не инициализирует ее настоящим массивом. Чтобы создать массив, нужно применить оператор *new* или прямую инициализацию:

```
тип[ ] имя_массива = new тип [размер] ;

тип имя_массива [ ] = new тип [размер] ;
```

После создания массива изменить его размер невозможно (можно изменять отдельные его элементы).

В Java выполняется автоматическая проверка факта выхода за пределы массива. Поэтому если в программном коде по ошибке выполняется обращение к несуществующему элементу массива, программа не скомпилируется. Размерность массива хранится в его свойстве *length*.

Многомерный массив – это массив массивов. Двухмерный массив – это ряд массивов. Чтобы объявить двухмерный массив необходимо использовать следующий синтаксис:

```
тип имя_массива [ ][ ] = new тип [размер1][размер2] ;
```

Общая форма объявления многомерного массива следующая:

```
тип имя_массива [ ]…[ ] = new тип [размер1]…[размерN] ;
```

Многомерный массив можно инициализировать, заключая инициализирующую последовательность для каждого размера массива в отдельные фигурные скобки следующим образом:

```
тип имя_массива [ ] … [ ] = {
   { val, val, …, val },
   …
   { val, val, …, val },
} ;
```

При выделении памяти для многомерного массива (т. е. при создании многомерного массива) достаточно указать лишь первый (крайний слева) размер. Для отдельных измерений массива не обязательно указывать одинаковое количество элементов для каждого измерения. Двухмерные массивы с разной размерностью второго измерения задаются в следующих примерах:

```
int twoD[ ][ ] = new int[4][ ];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];

int arr[ ][ ] = { { 1 },
                  { 2, 3 },
                  { 4, 5, 6 },
                  {7, 8, 9, 0 }
                };
```

Пример решения задачи

В программе пользователь вводит с клавиатуры несколько целых чисел через пробел, а затем подсчитывается сумма положительных чисел.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputSreamReader;

public class Example5 {
   public static void main(String[ ] args) throws IOException {
      //считываем числа с клавиатуры
      BufferedReader b =
         new BufferedReader(new InputSreamReader(System.in));
      String str = b.readLine( );
      b.close( );
      String[ ] num = str.split(" ");
      //массив целых чисел
      int [ ] n = new int[num.length];
      //подсчет суммы
      int sum = 0;
```

```
        for (int i = 0; i < n.length; i++) {
           if (n[i] > 0) sum += n[i];
        }
        System.out.println("Сумма положительных чисел " + sum);
    }
}
```

Задания для самостоятельной работы

Задание 5.1. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите слова в обратном порядке.

Задание 5.2. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Вычислите сумму чисел между первым и вторым положительными числами.

Задание 5.3. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите слова, в которых символы идут в строгом возрастании их кодов.

Задание 5.4. Введите с консоли несколько натуральных чисел. Числа вводятся в одну строку через пробел. Выведите все четные числа, а строкой ниже – нечетные числа.

Задание 5.5. Введите с консоли несколько натуральных чисел. Числа вводятся в одну строку через пробел. Найдите число, в котором количество различных цифр минимально. Если таких чисел несколько, то выведите их все. Выведите все простые числа среди введенных чисел.

Задание 5.6. Введите с консоли несколько натуральных чисел. Числа вводятся в одну строку через пробел. Выведите все трехзначные числа, в десятичной записи которых нет одинаковых цифр.

Задание 5.7. Введите с клавиатуры пароль и сравните его со строкой-образцом.

Задание 5.8. Введите с консоли несколько натуральных чисел. Числа вводятся в одну строку через пробел. Выведите все числа-палиндромы, значения которых в прямом и обратном порядке совпадают.

Задание 5.9. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите через запятую слова, длина которых строго больше шести.

Задание 5.10. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите все числа, равные полусумме соседних.

Задание 5.11. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите через пробел все слова, которые содержат цифры, например, a1 или abc3d.

Задание 5.12. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите слова, в которых одинаковое количество гласных и согласных букв.

Задание 5.13. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите все числа, которые делятся на 3 или на 5.

Задание 5.14. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите слова, состоящие только из различных символов.

Задание 5.15. Введите с консоли два натуральных числа. Определите их наибольший общий делитель (НОД).

Задание 5.16. Введите с консоли несколько русских и английских слов и целых чисел в одну строку через пробел. Выведите только русские слова.

Задание 5.17. Введите с консоли несколько целых чисел, причем числа могут повторяться. Числа вводятся в одну строку через пробел. Подсчитайте, сколько раз каждое число встречалось в строке.

Задание 5.18. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Выведите слова-палиндромы, которые можно читать в прямом и обратном порядках.

Задание 5.19. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите эти числа в порядке возрастания.

Задание 5.20. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Если слово содержит букву «р», то не выводите его; если слово содержит букву «л», то выведите его дважды; если слово содержит буквы «р» и «л», то выведите его один раз; остальные слова выведите один раз.

Задание 5.21. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите первые два положительных числа, расположенные подряд.

Задание 5.22. Введите с консоли несколько слов. Слова вводятся в одну строку через пробел. Если в слове четное число букв, слово выводится дважды; если в слове нечетное число букв, слово выводится один раз.

Задание 5.23. Введите с консоли несколько натуральных чисел. Числа вводятся в одну строку через пробел. Выведите числа, которые делятся на 5, а строкой ниже – числа, которые делятся на 7.

Задание 5.24. Введите с консоли несколько слов, причем слова могут повторяться. Слова вводятся в одну строку через пробел. Нужно подсчитать, сколько раз каждое слово встречалось в строке (регистр не учитывается).

Задание 5.25. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите для каждого числа его модуль.

Задание 5.26. Введите с консоли несколько символов в одну строку. Подсчитайте количество букв английского алфавита, которые есть в этой строке.

Задание 5.27. Введите с консоли несколько целых чисел. Числа вводятся в одну строку через пробел. Выведите минимальное и максимальное значения среди этих чисел.

Тема 6. КЛАСС ARRAYLIST

Основные теоретические сведения

В классе *ArrayList* поддерживаются динамические массивы, которые могут наращиваться по мере надобности. Стандартные массивы в Java имеют фиксированную длину. После того как массив создан, его размер не может быть изменен. Нужно заранее знать, сколько элементов нужно будет хранить, чтобы задать размер массива. Это не всегда известно. Можно использовать класс *ArrayList*. Класс *ArrayList* представляет собой списочный массив объектных ссылок переменной длины. Это означает, что размер объекта типа *ArrayList* может динамически увеличиваться или уменьшаться. Списочный массив создается с некоторым начальным размером. Когда же этого первоначального размера оказывается недостаточно, списочный массив автоматически расширяется. Когда из него удаляются объекты, он может сокращаться.

Класс *ArrayList* является обобщенным и объявляется приведенным ниже способом:

```
class ArrayList<E>
```

Параметр E обозначает тип сохраняемых объектов.
При этом параметром может быть только объектный тип.
Пример создания объекта класса *ArrayList* имеет следующий вид:

```
ArrayList<String> list = new ArrayList<String>( );
```

Некоторые основные методы класса представлены в таблице 6.

Таблица 6 – **Методы класса *ArrayList***

Метод	Описание
add(Object elem)	Добавляет элемент в конец списка
add(int index, Object elem)	Добавляет в список элемент на указанную позицию
remove(int index)	Удаляет элемент по переданному индексу
remove(Object elem)	Удаляет указанный элемент
contains(Object elem)	Возвращает *true*, если нашлось совпадение для переданного объекта
isEmpty( )	Возвращает *true*, если список не содержит элементов

Окончание таблицы 6

Метод	Описание
indexOf(Object elem)	Возвращает либо индекс объекта, переданного в параметре, либо –1
size( )	Возвращает количество элементов в списке на текущий момент
get(int index)	Возвращает объект, который сейчас находится по индексу, переданному в параметре
toArray( )	Преобразовывает список в массив

Пример решения задачи

В программе пользователь вводит с клавиатуры слова, которыми затем заполняется список *ArrayList*, пока не будет введено слово «end». Затем выводится самая длинная строка в списке (если таких строк несколько, то выводятся они все).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputSreamReader;
import java.util.ArrayList;

public class Example6 {
   public static void main(String[ ] args) throws IOException {
      //объявляем список строк
      ArrayList<String> list = new ArrayList( );
      //переменная max – длина самого длинного слова в списке
      int max = 1;
      //считываем строки с клавиатуры
      BufferedReader b =
         new BufferedReader(new InputSreamReader(System.in));
      String str = b.readLine( );
      while( ! str.equalsIgnoreCase("end")) {
         list.add(str);
         if (str.length > max) max = str.length;
         str = b.readLine( );
      }
      b.close( );

      //вывод самых длинных слов в списке
```

```
        System.out.println("Самые длинные слова в списке:");
        for (int i = 0; i < list.size( ); i++) {
           if (list.get(i) == max) System.out.println(list.get(i));
        }
     }
  }
```

Задания для самостоятельной работы

Задание 6.1. Считайте с консоли строки, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Определите самую короткую строку в списке. Если таких строк несколько, выведите их все.

Задание 6.2. Считайте с консоли строки, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*, но только добавляйте не в конец списка, а в начало. Определите самую длинную строку в списке. Если таких строк несколько, выведите их все.

Задание 6.3. Считайте с консоли строки, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*, добавляйте не в конец списка, а в начало. Определите самую короткую строку в списке. Если таких строк несколько, выведите их все.

Задание 6.4. Считайте с консоли строки, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Удалите третий элемент списка и выведите оставшиеся элементы.

Задание 6.5. Считайте с консоли строки, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Удалите второй элемент списка и выведите оставшиеся элементы.

Задание 6.6. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. После каждого слова вставьте слово «именно» и выведите список на консоль в одну строку через пробел.

Задание 6.7. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Удалите из списка все слова, содержащие букву «р» и выведите список на консоль в одну строку через пробел.

Задание 6.8. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Переставьте несколько первых слов в конец списка (число N вводится с клавиатуры). Выведите получившийся список на консоль в одну строку через пробел.

Задание 6.9. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Переставьте несколько последних слов в начало списка (число N вводится с клавиатуры). Выведите получившийся список на консоль в одну строку через пробел.

Задание 6.10. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. После каждого слова вставьте слово «Бум!» и выведите список на консоль в одну строку через пробел.

Задание 6.11. Считайте с консоли слова, вводимые пользователем, пока пользователь не введет слово «end». Заполните ими список *ArrayList*. Удалите из списка каждое второе слово. Выведите полученный список на консоль в одну строку через пробел.

Тема 7. ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Основные теоретические сведения

Основными принципами объектно-ориентированного программирования (ООП) являются следующие:

- инкапсуляция;
- полиморфизм;
- наследование.

Инкапсуляция представляет собой механизм программирования, объединяющий код и данные, которыми он манипулирует. Он предотвращает несанкционированный доступ к данным извне и их не-

корректное использование. В объектно-ориентированных языках программирования код и данные организуются в некое подобие «черного ящика». Инкапсуляцию можно считать защитной оболочкой, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося внутри оболочки. Сильная сторона инкапсулированного кода состоит в следующем: всем известно, как получить доступ к нему, следовательно, его можно использовать независимо от подробностей реализации и не опасаясь побочных эффектов.

Цель инкапсуляции – улучшить качество взаимодействия вещей за счет упрощения их. Лучший способ упростить что-то – это скрыть все сложное от посторонних глаз. С точки зрения программирования, инкапсуляция – это «сокрытие реализации». Основной языковой конструкцией, поддерживающей инкапсуляцию в Java, является класс.

Наследование – это процесс, в ходе которого один объект приобретает свойства другого объекта. С помощью наследования поддерживается иерархическая классификация.

Наследование также связано с инкапсуляцией. Если отдельный класс инкапсулирует определенные свойства, то любой его подкласс будет иметь те же самые свойства и некоторые дополнительные, определяющие его специализацию. Новый подкласс наследует атрибуты всех своих родительских классов и поэтому не содержит непредсказуемые взаимодействия с большей частью остального кода системы.

Полиморфизм – это свойство, позволяющее с помощью одного интерфейса обращаться к общему классу действий. Полиморфизм позволяет единообразно обращаться к объектам различных классов (обычно имеющих общего предка).

Модификаторы доступа (спецификаторы) (access specifiers) в Java следующие:

- *public* (открытый);
- *private* (закрытый);
- *protected* (защищенный);
- по умолчанию.

Модификатор доступа указывается перед остальной частью описания типа отдельного члена класса. Именно с него начинается объявление члена класса, например:

```
public int n;
private double j;
private int myMethod(int a, int b) { //…
```

К переменной, методу или классу, помеченному модификатором *public*, можно обращаться из любого места программы. Это самая высокая степень открытости. Никаких ограничений нет.

К переменной или методу, помеченному модификатором *private*, можно обращаться только из того же класса, где он объявлен. Для всех остальных классов помеченный метод или переменная невидимы и «как бы не существуют». Это самая высокая степень закрытости (только свой класс).

Класс нельзя объявить как *private* (это сделает класс недоступным для окружающих, использовать он сможет только «сам себя») или *protected*. Поэтому имеется такой выбор при задании доступа к классу (в пределах пакета или открытый (*public*). Если необходимо перекрыть доступ к классу для всех, то нужно объявить все его конструкторы со спецификатором *private*, соответственно, запретив кому бы то ни было создание объектов этого класса.

Доступ по умолчанию не имеет ключевого слова, но часто его называют доступом в пределах пакета (*package access*). Это значит, что член класса доступен для всех остальных классов текущего пакета, но для классов за пределами пакета он воспринимается как приватный (*private*). Так как компилируемый модуль (файл) может принадлежать лишь одному пакету, все классы одного компилируемого модуля автоматически открыты друг для друга в границах пакета.

Доступ в пределах пакета позволяет группировать взаимосвязанные классы в одном пакете, чтобы они могли легко взаимодействовать друг с другом. Размещая классы в одном пакете, код пакета берется под полный контроль. Таким образом, только принадлежащий вам код будет обладать пакетным доступом к другому, принадлежащему вам же коду. Это вполне логично. Можно сказать, что доступ в пределах пакета является основной причиной для группировки классов в пакетах. Во многих языках определения в классах организуются совершенно произвольным образом, но в Java более жесткая логика структуры. Вдобавок классы, которые не должны иметь доступ к классам текущего пакета, следует просто исключить из этого пакета.

К переменной, методу или классу, помеченному модификатором *protected*, можно обращаться из его же пакета (как *package*), но еще из всех классов, унаследованных от текущего.

Геттер (от англ. getter – получатель) – специальный метод в программировании, позволяющий получить данные, доступ к которым напрямую ограничен. Это один из методов объектно-ориентированного программирования, который помогает реализовать гибкий механизм инкапсуляции.

Сеттер (от англ. setter) – метод, используемый для того, чтобы присвоить какое-либо значение инкапсулированному полю, например, обработав при этом недопустимые присваивания.

Перегрузка методов является одним из способов реализации принципа полиморфизма в Java. Метод называется *перегруженным*, если существует несколько его версий с одним и тем же именем, но с различными типами и (или) списком параметров. Для того чтобы перегрузить метод, достаточно объявить его новый вариант, отличающийся от уже существующих, соблюдая условие (тип и (или) число параметров в каждом из перегружаемых методов должны быть разными). При этом одного лишь различия в типах возвращаемых значений недостаточно.

Перегрузка может ограничиваться одним классом.

Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными. Если в последнем случае списки параметров совпадают, то имеет место другой механизм (*переопределение* метода).

Перегрузка реализует «раннее связывание».

Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами.

Задания для самостоятельной работы

Задание 7.1. Исправьте ошибку в программе, чтобы переменная *age* объекта *person* изменила свое значение. Ошибка допущена в методе *adjustAge*.

Текст программы имеет следующий вид:

```
public class Exercise7_1 {
   public static void main(String[ ] args) {
      Person person = new Person( );
      System.out.println("Age is: " + person.age);
      person.adjustAge(person.age);
      System.out.println("Adjusted Age is: " + person.age);
   }

   public static class Person {
      public int age = 20;

      public void adjustAge(int age) {
```

```
            age = age + 10;
            System.out.println("The Age in adjustAge( ) is " + age);
        }
    }
}
```

Задание 7.2. В папке *theme7* создайте папку (package) с именем *exercise7_2*. В папке *exercise7_2* создайте класс *Cat* (кот) со следующими конструкторами:

- имя;
- имя, вес, возраст;
- имя, возраст (вес стандартный);
- вес, цвет (имя, адрес и возраст – неизвестные, кот – бездомный);
- вес, цвет, адрес (чужой домашний кот).

Задача конструктора – сделать объект валидным. Например, если вес не известен, то нужно указать какой-нибудь средний вес. Кот не может ничего не весить. То же относительно возраста. Имени или адреса может и не быть (*null*).

Все переменные класса *Cat* сделайте *private*.

В классе *Cat* создайте перегруженный метод *toString*( ) для вывода на консоль сведений о коте.

В папке *exercise7_2* создайте класс *CatTest*. В классе *CatTest* создайте метод *main*. В методе *main* создайте несколько котов (объектов класса *Cat*), используя разные конструкторы. Выведите сведения о каждом коте.

Задание 7.3. В папке *theme7* создайте папку (package) с именем *exercise7_3*. В папке *exercise7_3* создайте классы *GoodDog* и *GoodDogTestDrive* на основании следующего примера:

```
class GoodDog {
   private int size;
   public int getSige( ) {
      return size;
   }
   public setSize(int size) {
      this.size = size;
   }
   void bark() {
      if (size > 60) System.out.println("Гав! Гав!");
      else if (size > 14) System.out.println("Вуф! Вуф!");
```

```
      else System.out.println("Тяф! Тяф!");
   }
}

class GoodDogTestDrive {
   public static void main(String[ ] args) {
      GoodDog one = new GoodDog();
      one.setSize(70);
      GoodDog two = new GoodDog();
      two.setSize(8);
      System.out.println("Первая собака: " + one.getSize());
      System.out.println("Вторая собака: " + two.getSize());
      one.bark();
      two.bark();
   }
}
```

Задание 7.4. Задание выполняется на основе задания 3.6. В папке *theme7* создайте папку (package) с именем *exercise7_4*. В папке *exercise7_4* создайте два класса как в задании 3.6. В первом классе все переменные объявите *private*. В первом классе создайте два конструктора. Первый конструктор создайте как в задании 3.6. Второй конструктор должен содержать меньше параметров, чем первый. Оставшиеся переменные должны инициализироваться некоторыми константами. Во всех конструкторах необходимо использовать ключевое слово *this*. При создании конструктора в IntelliJ Idea используйте команду Alt + Insert, а затем в списке нужно выбрать конструктор и переменные для инициализации в создаваемом конструкторе.

Для всех переменных первого класса создайте геттеры и сеттеры.

Создайте в первом классе метод как в задании 3.6.

Создайте второй класс как в задании 3.6.

Тема 8. ПРИМЕНЕНИЕ КЛЮЧЕВОГО СЛОВА STATIC

Основные теоретические сведения

Когда описываются переменные в классе, указывается, будут ли эти переменные созданы всего один раз или же нужно создавать их копии для каждого объекта. По умолчанию создается новая копия пе-

ременной для каждого объекта. Иногда требуется определить такой член класса, который будет использоваться независимо от каких бы то ни было объектов этого класса. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово *static*. Если член класса объявляется как *static*, он становится доступным до создания каких-либо объектов своего класса и без ссылки на какой-либо объект.

Статические переменные существуют в одном экземпляре, обращаться к ним нужно по имени класса. Для того чтобы воспользоваться членом типа *static* за пределами класса, достаточно дополнить имя данного члена именем класса, используя точечную нотацию. Создавать объект для этого не нужно. В действительности член типа *static* оказывается доступным не по ссылке на объект, а по имени своего класса.

Если требуется присвоить значение 10 переменной *count* типа *static*, являющейся членом класса *Timer*, то для этой цели можно воспользоваться следующей строкой кода:

```
Timer.count = 10;
```

Эта форма записи подобна той, которая используется для доступа к обычным переменным экземпляра посредством объекта, но в ней указывается имя класса, а не объекта.

Переменные, объявляемые как *static*, по существу являются глобальными. В силу этого при создании объектов данного класса копии статических переменных в них не создаются. Вместо этого все экземпляры класса совместно пользуются одной и той же статической переменной.

С помощью ключевого слова *static* можно объявлять не только переменные, но и методы, например:

```
public static void myMethod ( ) { //…
```

Метод типа *static* отличается от обычного метода тем, что его можно вызывать по имени его класса, не создавая экземпляр объекта этого класса, например:

```
MyClass. myMethod ( );
```

Ключевое слово *static* позволяет методу работать без экземпляра класса. Статический метод подразумевает, что его поведение не зави-

сит от переменных экземпляра, поэтому нет необходимости в экземпляре (объекте). Нужен просто класс.

Обычные методы вызываются у объекта и имеют доступ к данным этого объекта. Статические методы не имеют такого доступа. У них просто нет ссылки на объект, они способны обращаться к статическим переменным этого класса либо к другим статическим методам.

Статические методы не могут обращаться к нестатическим методам или нестатическим переменным.

Каждая обычная переменная класса находится внутри объекта. Обратиться к ней можно только имея ссылку на этот объект. В статический метод такая ссылка не передается.

В каждый обычный метод неявно передается ссылка на объект, у которого этот метод вызывают. Переменная, которая хранит эту ссылку, называется *this*. Метод всегда может получить данные из своего объекта или вызвать другой нестатический метод этого же объекта.

В статический метод вместо ссылки на объект передается *null*. Поэтому он не может обращаться к нестатическим переменным и методам. У него просто нет ссылки на объект, к которому они привязаны.

На методы, объявленные как *static*, накладываются следующие ограничения:

- они могут непосредственно вызывать только другие статические методы;
- им непосредственно доступны только статические переменные;
- они не могут делать ссылки типа *this* или *super*.

Нестатический метод всегда может вызвать статический метод из своего класса или получить доступ к статической переменной.

Иногда для подготовки к созданию объектов в классе должны быть выполнены некоторые инициализирующие действия. В частности, может возникнуть потребность установить соединение с удаленным сетевым узлом или задать значения некоторых статических переменных перед тем, как воспользоваться статическими методами класса.

Для решения подобных задач в Java предусмотрены следующие *статические блоки*:

```
static {
   …
}
```

Статический блок выполняется при первой загрузке класса, еще до того, как класс будет использован для каких-нибудь других целей.

Класс *java.lang.Math* – замечательный пример, в котором почти все методы статичны. Класс имеет следующие переменные, которые также являются статическими:

```
public static final double E;
public static final double PI;
```

Задания для самостоятельной работы

Задание 8.1. В папке *theme8* создайте папку (package) с именем *exercise8_1*. В папке *exercise8_1* создайте класс *Cat*. В классе *Cat* создайте статическую переменную *ArrayList<Cat> cats*. В классе *Cat* создайте метод *main*. В методе *main* создайте 10 объектов класса *Cat*. При каждом создании кота (нового объекта класса *Cat*) в переменную *cats* добавляется этот новый кот.

В классе *Cat* создайте статический метод *printCats*( ). Метод должен выводить всех котов на консоль. Метод должен использовать переменную *cats*. В методе *main* вызовите метод *printCats*( ).

Задание 8.2. В нижеприведенном тексте программы переставьте один модификатор *static*, чтобы она скомпилировалась:

```
public class Exercise8_2 {
   public int A = 5;
   public int B = 2;
   public static int C = A*B;

public static void main(String[ ] args) {
      A = 15;
   }
}
```

Задание 8.3. В нижеприведенном тексте программы расставьте минимальное количество слов *static*, чтобы код начал работать и программа успешно завершилась:

```
public class Solution {
   public int step;

public static void main(String[ ] args) {
      method1( );
   }
```

```
public void method1( ) {
     method2( );
   }
public void method2( ) {
     new Solution( ).method3();
   }

public void method3( ) {
     method4( );
   }

public void method4( ) {
     step++;
     for (StackTraceElement element :
          Thread.currentThread().getStackTrace())
        System.out.println(element);
     if (step > 1) return;
     main(null);
   }
}
```

Задание 8.4. В нижеприведенном тексте программы укажите, что в какой последовательности инициализируется. Исправьте порядок инициализации данных так, чтобы результат был следующим:

```
static void init( )
Static block
public static void main
non-static block
static void printAllFields
0
null
Solution constructor
static void printAllFields
6
First name
```

Текст программы имеет следующий вид:

```
public class Solution {
   static {
```

```
      System.out.println("Static block");
   }

{
      System.out.println("non-static block");
      printAllFields(this);
   }

public int i = 6;

public String name = "First name";
   static {
      init();
   }

public Solution() {
      System.out.println("Solution constructor");
      printAllFields(this);
   }

public static void init() {
      System.out.println("static void init()");
   }

public static void main(String[ ] args) {
      System.out.println("public static void main");
      Solution s = new Solution();
   }

public static void printAllFields(Solution obj) {
      System.out.println("static void printAllFields");
      System.out.println(obj.name);
      System.out.println(obj.i);
   }
}
```

Тема 9. НАСЛЕДОВАНИЕ

Основные теоретические сведения

Одним из трех фундаментальных принципов объектно-ориентированного программирования является наследование – механизм, обеспечивающий создание иерархических классификаций. Используя наследование, можно создать общий класс, определяющий характеристики, которые будут общими для множества родственных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять собственные уникальные характеристики.

В терминологии Java общий класс называют *суперклассом*, а наследующий – *подклассом*. Подкласс – это специализированная версия суперкласса. Он наследует все переменные и методы, определенные в суперклассе, дополняя их собственными, уникальными элементами.

В Java чтобы наследовать класс, достаточно включить его имя в объявление другого класса с помощью ключевого слова *extends*, например:

```
class Cat extends Animal {
…
}
```

К правилам, позволяющим правильно сформировать иерархию классов, т. е. наследование, относятся следующее:

- Используйте наследование для класса, который представляет собой более специфичный вид родительского класса. Например, *Willow* (ива) – это тип *Tree* (дерево), поэтому *Willow* расширяет *Tree*.
- Применяйте наследование, если у вас есть поведение (реализованный код), которое должно быть общим для разных классов. Например, *Square* (квадрат), *Circle* (круг) и *Triangle* (треугольник) – подклассы класса *Shape* (фигура). Это также улучшит сопровождаемость и расширяемость кода. Наследование – одна их ключевых возможностей объектно-ориентированного программирования, но оно не всегда должно рассматриваться как наилучший способ повторного использования кода.
- Не применяйте наследование только для того, чтобы иметь возможность повторно использовать код из других классов, если отношения между родительским и дочерним классами нарушают хотя бы одно из двух вышеперечисленных правил.

• Не используйте наследование, если дочерний и родительский классы не проходят проверку на соответствие. Всегда спрашивайте себя, является ли дочерний класс частным случаем родительского класса. Например, выражение «чай – это напиток» имеет смысл, а «напиток – это чай» – нет.

Когда объявляется общий тип для группы классов, то можно заменять родительский класс дочерним везде, где предполагается его присутствие. Полиморфизм допускает, что ссылка и объект могут иметь разные типы, например:

```
Animal myDog = new Dog( );
```

Создав суперкласс, в котором определены общие для множества объектов свойства, можно использовать его для создания любого числа более специализированных подклассов. Каждый подкласс добавляет собственный набор специфических для него атрибутов в соответствии с конкретной необходимостью.

С целью исключения несанкционированного доступа к членам класса их часто объявляют как закрытые, используя для этого модификатор доступа *private*. Наследование класса не отменяет ограничений, налагаемых на доступ к закрытым членам класса. Несмотря на то что в подкласс автоматически включаются все члены его суперкласса, доступ к закрытым членам суперкласса ему запрещен.

В иерархии классов допускается, чтобы как суперклассы, так и подклассы имели собственные конструкторы.

В связи с этим возникает следующий вопрос: какой именно конструктор отвечает за создание объекта подкласса (конструктор суперкласса, конструктор подкласса или же оба одновременно). Конструктор суперкласса используется для построения родительской части объекта, конструктор подкласса – для остальной его части.

В этом есть своя логика, поскольку суперклассу неизвестны и недоступны любые собственные члены подкласса, а значит, каждая из указанных частей объекта должна конструироваться по отдельности.

Если конструктор определен только в подклассе, то все происходит очень просто. Конструируется объект подкласса, а родительская часть объекта автоматически создается конструктором суперкласса, используемым по умолчанию.

Для вызова конструктора суперкласса из подкласса используется следующая общая форма ключевого слова *super*:

```
super (список_ параметров) ;
```

Список_ параметров определяет параметры, которые нужны конструктору суперкласса.

Вызов конструктора *super*( ) всегда должен быть первым оператором в теле конструктора подкласса.

Если в суперклассе имеется несколько перегруженных конструкторов, то вызов *super*( ) позволяет вызвать любую форму конструктора, определенную в суперклассе. Для выполнения выбирается тот вариант конструктора, который соответствует указанным аргументам.

Существует еще одна общая форма ключевого слова *super*, которая применяется подобно ключевому слову *this*, но ссылается на суперкласс данного класса. Эта общая форма обращения к члену суперкласса имеет следующий вид:

```
super.член_класса
```

Член_класса обозначает метод или переменную экземпляра.

В отношении наследования и иерархии классов может возникнуть следующий вопрос: когда именно создается объект подкласса и какой именно конструктор выполняется первым (тот, который определен в подклассе, или же тот, который определен в суперклассе). Например, если имеется суперкласс A и подкласс B, то необходимо определить, что вызывается раньше: конструктор класса A или конструктор класса B.

В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Более того, метод *super*( ) должен быть первым оператором в конструкторе подкласса, и поэтому порядок, в котором вызываются конструкторы, остается неизменным, независимо от того, используется ли вызов *super*( ) или нет. Если вызов *super*( ) отсутствует, то выполняется конструктор каждого суперкласса по умолчанию (т. е. конструктор без параметров).

Вызов конструкторов в порядке наследования классов имеет определенный смысл. Ведь суперклассу ничего не известно ни об одном из производных от него подклассов, поэтому любая инициализация, которая требуется его членам, не только должна осуществляться независимо от инициализации членов подкласса, но и, возможно, является необходимой подготовительной операцией для этого процесса. Следовательно, она должна выполняться первой.

В иерархии классов часто присутствуют методы с одинаковой сигнатурой и одинаковым возвращаемым значением как в суперклассе,

так и в подклассе. В этом случае говорят, что метод суперкласса переопределяется в подклассе.

Если переопределяемый метод вызывается из подкласса, то он всегда будет ссылаться на версию метода, определенную в подклассе. Версия метода, определенная в суперклассе, скрывается.

Ссылочная переменная суперкласса может ссылаться на объект подкласса. В Java этот принцип используется для вызова переопределяемых методов во время выполнения.

Если вызов переопределенного метода осуществляется с использованием ссылки на суперкласс, то исполняющая система Java выбирает нужную версию метода на основании типа объекта, на который эта ссылка указывает в момент вызова. Ссылкам на различные типы объектов будут соответствовать вызовы различных версий переопределенного метода.

Во время выполнения версия переопределенного метода выбирается в зависимости от *типа объекта ссылки* (не типа ссылочной переменной).

Следовательно, если суперкласс содержит метод, переопределенный в подклассе, будет вызываться метод, соответствующий тому объекту, на который указывает ссылочная переменная суперкласса.

Переопределяемые методы обеспечивают поддержку полиморфизма времени выполнения.

Многоуровневая иерархия классов содержит класс, его родитель, родитель родителя и т. д. до самого класса *Object*. Если в результате присваивания происходит движение по цепочке наследования вверх (к типу *Object*), то это *сужение* типа, а если вниз, к типу объекта, то это *расширение* типа.

Движение вверх по цепочке наследования называется сужением, так как теряется возможность вызвать методы, которые были добавлены в класс при наследовании.

При расширении типа нужно использовать *оператор преобразования типа*. При этом Java-машина выполняет проверку, а действительно ли данный объект унаследован от типа, к которому мы хотим его преобразовать.

Задания для самостоятельной работы

Задание 9.1. В папке *theme9* создайте папку (package) с именем *theme9_1*. В папке *theme9_1* создайте класс *Employee* (сотрудник). В классе *Employee* создайте приватную переменную *id* типа *int* и конструктор, позволяющий инициализировать переменную *id*.

В классе *Employee* создайте геттер.

В классе *Employee* создайте метод *typeEmployee*( ) без параметров, который выводит на консоль слово «Сотрудник». Подумайте, какой должен быть возвращаемый тип у этого метода.

В папке *theme9_1* создайте класс *Manager* (менеджер). Унаследуйте менеджера от сотрудника. В классе *Manager* создайте приватную переменную *idProject* типа *int*.

В классе *Manager* создайте конструктор с двумя параметрами, позволяющий инициализировать переменную подкласса *idProject* и переменную суперкласса *id*. Для инициализации переменной суперкласса предусмотрите вызов конструктора суперкласса.

В классе *Manager* создайте геттер.

В классе *Manager* создайте метод *typeEmployee*( ) без параметров, который выводит на консоль слово «Менеджер». Подумайте, какой должен быть возвращаемый тип у этого метода.

В папке *theme9_1* создайте класс *EmployeeDemo* с методом *main*. В методе *main* введите следующее:

```
Employee b1 = new Employee(7110);
Employee b2 = new Manager(9251, 31);
b1.typeEmployee(); //
b2.typeEmployee(); //
b2.getIdProject();
Manager b3 = new Manager(9711, 35);
System.out.println(b3.getIdProject()); //
System.out.println(b3.getId()); //
```

Исправьте ошибку в методе *main*. В тех местах программы, которые оставлены для комментариев, впишите результат выполнения соответствующих строчек.

Задание 9.2. Напишите реализацию метода, который определяет, объект какого класса ему передали, возвращает результат (одно из значений «Корова», «Кит», «Собака», «Неизвестное животное»).

Текст программы имеет следующий вид:

```
public class Exercise9_2 {
   public static void main(String[ ] args) {
      System.out.println(getObjectType(new Cow( )));
      System.out.println(getObjectType(new Dog( )));
      System.out.println(getObjectType(new Whale( )));
```

```
        System.out.println(getObjectType(new Pig( )));
    }

    public static String getObjectType(Object o) {
        //Напишите тут ваше решение
        return "Неизвестное животное";
    }

    public static class Cow { }

    public static class Dog { }

    public static class Whale { }

    public static class Pig { }
}
```

Задание 9.3. В папке *theme9* создайте папку (package) с именем *theme9_3*. В папке *theme9_3* создайте три класса: *Pet* (домашнее животное), *Cat* (кот) и *Dog* (собака). Унаследуйте кота и собаку от домашнего животного.

В классах *Cat* и *Dog* создайте методы *makeNoise*( ), которые выводят на консоль «Мяу!» или «Гав!» соответственно. Подумайте, какой должен быть возвращаемый тип у этого метода.

В папке *theme9_3* создайте класс *PetDemo* с методом *main*. В методе *main* создайте переменные *pet1*, *pet2*, *pet3*, *pet4* типа *Pet*. Присвойте этим переменным ссылки на объекты разных типов. У переменных *pet1*, *pet2*, *pet3*, *pet4* вызовите метод *makeNoise*( ).

Тема 10. АБСТРАКТНЫЕ КЛАССЫ, АБСТРАКТНЫЕ МЕТОДЫ, ИНТЕРФЕЙСЫ

Основные теоретические сведения

Иногда требуется создать суперкласс, в котором определяется лишь самая общая форма для всех его подклассов, а наполнение ее деталями предоставляется каждому из этих подклассов. В таком суперклассе определяется лишь суть методов, которые должны быть конкретно реализованы в подклассах, а не в самом суперклассе. Подобная ситуация возникает, например, в связи с невозможностью

полноценной реализации метода в суперклассе. В подобных случаях требуется какой-то способ, гарантирующий, что в подклассе действительно будут переопределены все необходимые методы. Такой способ есть в Java. Он состоит в использовании *абстрактного метода*.

Абстрактный метод объявляется с использованием модификатора *abstract*. Абстрактный метод не имеет тела и поэтому не реализуется в суперклассе. Это означает, что он должен быть переопределен в подклассе, поскольку его вариант из суперкласса просто непригоден для использования. Для определения абстрактного метода используется следующий общий синтаксис:

```
abstract тип имя(список_параметров);
```

В этом синтаксисе отсутствует тело метода.

Спецификатор *abstract* может применяться только к обычным методам, но не к статическим методам и конструкторам.

Если в классе есть хотя бы один абстрактный метод, то класс также должен быть помечен модификатором *abstract*. В абстрактном классе могут быть не только абстрактные, но и неабстрактные методы.

Интерфейсы аналогичны абстрактным классам. При этом возможно унаследовать класс от нескольких интерфейсов.

Задания для самостоятельной работы

Задание 10.1. В папке *theme10* создайте папку (package) *theme10_1*. В папке *theme10_1* создайте интерфейс *RepkaItem*. В интерфейсе *RepkaItem* создайте открытый метод *getNamePadezh*( ) типа *String*.

В папке *theme10_1* создайте открытый класс *Person* (персонаж сказки «Репка»). Класс *Person* реализует интерфейс *RepkaItem*. Создайте две приватные переменные *name* и *namePadezh* типа *String*. Создайте открытый конструктор с двумя параметрами для инициализации переменных класса. Для переменной *namePadezh* создайте геттер.

В классе *Person* создайте открытый метод *pull*( ) типа *void* с параметром типа *Person*. Метод должен выводить на консоль следующую фразу: «<имя> за <имя другого персонажа в винительном падеже>». Например, «Бабка за Дедку».

В папке *theme10_1* создайте открытый класс *RepkaStory*. В классе *RepkaStory* создайте статический метод *tell*( ), параметром которого является список персонажей типа *Person* (т. е. *ArrayList<Person>*),

возвращаемый тип метода – *void*. В метод *tell*( ) для всех последовательных пар персонажей должен вызываться метод *pull*( ).

В папке *theme10_1* создайте открытый класс *Exercise10_1*. В классе *Exercise10_1* создайте метод *main*, в котором создайте список персонажей сказки, имя списка – *plot*. В список *plot* добавьте следующих персонажей:

```
Person("Репка", "Репку")
Person("Дедка", "Дедку")
Person("Бабка", "Бабку")
Person("Внучка", "Внучку")
```

В методе *main* вызовите метод *tell*( ). В результате выполнения программы вывод на консоль должен быть следующим:

```
Внучка за Бабку
Бабка за Дедку
Дедка за Репку
```

Добавьте в программу остальных персонажей сказки «Репка».

Задание 10.2. В папке *theme10* создайте папку (package) *theme10_2*. В папке *theme10_3* создайте три класса. Правильно расставьте наследование между классами. Заполните пропуски, чтобы вывод программы был следующий: «Дрейф! Дрейф! Поднять паруса!»

Текст программы имеет следующй вид:

```
public class Rowboat __________ __________ {
   public __________ rowTheBoat( ) {
      System.out.print("Давай, Наташа!");
   }
}

public class __________ {
   private int __________ ;
   __________ void __________ (__________ __________ ) {
      length = len;
   }
   public __________ move( ) {
      System.out.print("__________ ");
   }
}
```

```
public class TestBoats {
   __________ __________ __________ main(String[ ] args) {
      __________ b1 = new Boat( );
      Sailboat b2 = new __________ ( );
      Rowboat __________ = new Rowboat( );
      b2.setLength(32);
      b1. __________ ( );
      b3. __________ ( );
      __________ .move( );
   }
}

public class __________ __________ Boat {
   public __________ __________ ( ) {
      System.out.print("__________ ");
   }
}
```

Тема 11. ВЛОЖЕННЫЕ И ВНУТРЕННИЕ КЛАССЫ

Основные теоретические сведения

В языке Java допускается определять один класс в другом классе. Такой класс называют вложенным (nested class), например:

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```

Область действия вложенного класса ограничена областью действия внешнего класса. Если класс B определен в классе A, то класс B не может существовать независимо от класса A.

Вложенный класс имеет доступ к членам (в том числе закрытым) того класса, в который он вложен. Вложенный класс, объявленный непосредственно в области действия своего внешнего класса, считается его членом.

Вложенный класс может быть объявлен как *public*, *private*, *protected* или *package default*.

Можно также объявлять вложенные классы, являющиеся локальными для блока кода.

Существуют два типа вложенных классов: статические и нестатические. Вложенный класс, объявленный с модификатором *static*, называется статическим вложенным классом (*static nested class*). Вложенный нестатический класс называется внутренним классом (*inner class*).

В Java возможность определить (вложить) один класс внутри определения другого класса позволяет группировать классы, логически связанные друг с другом, динамично управлять доступом к ним. С одной стороны, обоснованное использование в коде внутренних классов делает его более эффектным и понятным. С другой стороны, применение внутренних классов является одним из способов сокрытия кода, так как внутренний класс может быть абсолютно недоступен и не виден вне класса-владельца.

Одной из важнейших причин использования внутренних классов является возможность независимого наследования внутренними классами. Фактически при этом реализуется множественное наследование со своими преимуществами и проблемами.

Нестатические вложенные классы принято называть внутренними (*inner*) классами. При объявлении внутреннего класса могут использоваться модификаторы *final*, *abstract*, *private*, *protected*, *public*.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, могут непосредственно ссылаться на них таким же образом, как это делают остальные нестатические члены внешнего класса. Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим. Вложенный статический класс логически связан с классом-владельцем, но может быть использован независимо от него. При объявлении такого вложенного статического класса присутствует служебное слово *static*. Слово *static* перед объявлением вложенного класса указывает, что этот класс не хранит в себе ссылок на объекты внешнего класса, внутри которого объявлен, в этом вложенный ста-

тический класс подобен статическим методам. Если класс вложен в интерфейс, то он становится статическим по умолчанию.

Вложенный статический класс способен наследовать другие классы, реализовывать интерфейсы, являться объектом наследования для любого класса, обладающего необходимыми правами доступа.

Вложенный статический класс напрямую имеет доступ только к статическим полям и методам внешнего класса. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса.

Для создания объекта вложенного класса объект внешнего класса нет необходимости создавать.

Подкласс вложенного статического класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.

Локальные классы – это классы, объявленные в блоке операторов между фигурными скобками. На практике чаще всего объявление происходит в методе некоторого другого класса. Хотя объявлять локальный класс можно внутри статических и нестатических блоков инициализации.

Локальный класс имеет доступ к членам класса, в котором он объявлен. Локальный класс имеет доступ к локальным переменным. Локальные классы имеют доступ только к переменным, объявленным как *final*. Когда локальный класс обращается к локальной переменной, происходит «захват» этой переменой. Начиная с Java SE 8, локальные классы имеют доступ к финальным (*final*) локальным переменным и параметрам, а также к неизменяемым (*effectively final*) переменным, т. е. к переменным, которые не изменились с момента инициализации. Начиная с Java SE 8, локальный класс в теле метода имеет доступ к параметрам этого метода.

Локальные классы похожи на внутренние. Они не могут содержать статические поля или методы. Локальные классы в статических методах могут обращаться только к статическим полям внешнего класса. Локальный класс может содержать статические поля, только если они являются константами. Нельзя объявить интерфейс внутри метода, интерфейсы статичны.

У локальных классов есть следующие ограничения:

- они видны только в пределах блока, в котором объявлены;
- они не могут быть объявлены как *private*, *public*, *protected* или *static*;
- они не могут иметь внутри себя статических объявлений (полей, методов, классов); исключением являются константы (*static final*).

Анонимные классы позволяют сделать код более кратким. Анонимные классы позволяют одновременно объявить класс и создать его экземпляр. Анонимные классы, в отличие от локальных, не имеют имени. Используйте анонимные классы, если локальный класс нужен лишь один раз. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора *new*.

Определение анонимного класса содержит следующее:

- Оператор *new*.
- Имя интерфейса для реализации или класса для наследования.
- Круглые скобки, которые содержат аргументы для конструктора, также как и при создании обычных объектов. Если анонимным классом реализуется интерфейс, то скобки пустые.
- Тело класса.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области (или одноразового) применения метода.

Конструкторы анонимных классов нельзя определять и переопределять.

Анонимные классы допускают вложенность друг в друга.

Использование анонимных классов оправдано в следующих случаях:

- тело класса является очень коротким;
- нужен только один экземпляр класса;
- класс используется в месте его создания или сразу после него;
- имя класса не важно и не облегчает понимание кода.

Задание для самостоятельной работы

В папке *theme11* создайте абстрактный класс *Hen* (курица). В классе *Hen* создайте абстрактный метод *getCountOfEggsPerMonth*( ) с возвращаемым типом *int*. Создайте метод *getDescription*( ), который возвращает строку «Я курица.»

В папке *theme11* создайте классы *RussianHen*, *UkrainianHen*, *BelarusianHen*, *GermanHen*, которые наследуются от класса *Hen*. В каждом из четырех последних классов напишите свою реализацию метода *getCountOfEggsPerMonth*( ). Метод должен возвращать количество яиц в месяц от данного типа кур. В каждом из четырех последних классов напишите свою реализацию метода *getDescription*( ),

который возвращает строку следующего вида: *getDescription*( ) родительского класса, затем следующий вывод: «Моя страна – … Я несу N яиц в месяц.»

В папке *theme11* создайте интерфейс *Country*. В интерфейсе *Country* создайте следующие переменные типа *String*: *UKRAINE, RUSSIA, BELARUS, GERMANY*. Инициализируйте эти переменные.

В папке *theme11* создайте класс *Exercise11*. В классе *Exercise11* создайте статический класс *HenFactory*. В классе *HenFactory* создайте статический метод *getHen*( ) с параметром – название страны типа *String*. Метод возвращает породу кур, соответствующую стране, которая указана в интерфейсе *Country*. Подумайте, какой должен быть возвращаемый тип у этого метода.

В классе *Exercise11* создайте метод *main*. В методе *main* создайте по одному экземпляру каждого класса-потомка класса *Hen*, используя метод *getHen*( ). Для каждого созданного экземпляра вызовите метод *getDescription*( ).

Тема 12. ВВЕДЕНИЕ В ОБРАБОТКУ ИСКЛЮЧЕНИЙ

12.1. Обработка исключений в блоке *try/catch/finally*

Нередко в процессе выполнения программы могут возникать ошибки, притом необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Так, например, может неожиданно оборваться сетевое подключение при передаче файла; пользователь может ввести недопустимые данные; файл, который необходимо открыть, не найден. Подобные ситуации называются исключениями.

Исключение в Java – это объект некоторого класса, который описывает исключительное состояние, возникшее в каком-либо участке программного кода.

При возникновении исключения исполняющая система Java создает объект класса, связанного с данным исключением. Этот объект хранит информацию о возникшей исключительной ситуации (точка возникновения, описание и т. п.).

Возможна как автоматическая, так и программная генерация исключений.

Для обработки исключений используются ключевые слова *try, catch, finally, throw* и *throws*.

Блок *try/catch* размещается в начале и конце кода, который может сгенерировать исключение. Код в составе блока *try/catch* является защищенным кодом, синтаксис использования *try/catch* выглядит следующим образом:

```
try {
   // Защищенный код
}catch(НазваниеИсключения e1) {
   // Блок catch
}
```

Код, предрасположенный к исключениям, размещается в блоке *try*. В случае возникновения исключения обработка данного исключения будет производиться соответствующим блоком *catch*. За каждым блоком *try* должен немедленно следовать блок *catch* либо блок *finally*.

Оператор *catch* включает объявление типа исключения, которое предстоит обработать. При возникновении исключения в защищенном коде, блок *catch* (либо блоки), следующий за *try*, будет проверен. Если тип произошедшего исключения представлен в блоке *catch*, исключение передается в блок *catch* аналогично тому, как аргумент передается в параметр метода.

Ниже представлен массив с заявленными двумя элементами. Попытка кода получить доступ к элементу массива с индексом 3 повлечет за собой генерацию исключения.

```
import java.io.*;

public class Test {

   public static void main(String args[ ]) {
      System.out.println("До блока try-catch");
      try {
         int array[] = new int[2];
         System.out.println("Доступ к элементу:" + array[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Исключение:" + e);
      }
      System.out.println("После блока try-catch");
   }
}
```

Вследствие этого будет получен следующий результат:

```
До блока try-catch
Исключение: java.lang.ArrayIndexOutOfBoundsException: 3
После блока try-catch
```

За блоком *try* могут следовать несколько блоков *catch*. Синтаксис многократных блоков *catch* выглядит следующим образом:

```
try {
   // Защищенный код
}catch(ИсключениеТип1 e1) {
   // Блок catch
}catch(ИсключениеТип2 e2) {
   // Блок catch
// …
}catch(ИсключениеТипN eN) {
   // Блок catch
}
```

Сначала выполняется код, заключенный в фигурные скобки оператора *try*. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается за закрывающую фигурную скобку последнего оператора *catch*, ассоциированного с данным оператором *try*. Когда генерируется исключение, каждый оператор *catch* проверяется по порядку, выполняется тот из них, который совпадает по типу с возникшим исключением. В случае возникновения исключения в защищенном коде, исключение выводится в первый блок *catch* в списке. Если тип данных генерируемого исключения совпадает с *ИсключениеТип*1, он перехватывается в указанной области. В обратном случае, исключение переходит ко второму оператору *catch*. Это продолжается до тех пор, пока не будет произведен перехват исключения. По завершении одного из операторов *catch* все остальные пропускаются, выполнение программы продолжается с оператора, следующего сразу после блока операторов *try/catch*. Если исключение прошло через все операторы *catch*, но подходящего типа не обнаружено, выполнение текущего метода будет прекращено, исключение будет перенесено к предшествующему методу в стеке вызовов. Если возникла исключительная ситуация, класс которой не указан в качестве аргумента ни в одном *catch*, то выполнение всего *try* завершается нештатно.

Используя различные блоки *catch*, можно разграничить обработку различных типов исключений.

Текст программы имеет следующий вид:

```
int[] numbers = new int[3];
try{
   numbers[6] = 45;
   numbers[6] = Integer.parseInt("gfd");
}
catch(ArrayIndexOutOfBoundsException ex){

   System.out.println("Выход за пределы массива");
}
catch(NumberFormatException ex){
   System.out.println("Ошибка преобразования из строки в
число");
}
```

В среде Java 7 можно произвести обработку более чем одного исключения при использовании одного блока *catch*, что упрощает код, например:

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
}
```

Ключевое слово *finally* следует за блоком *try* либо блоком *catch*. Блок *finally* в коде выполняется всегда независимо от наличия исключения. Синтаксис блока *finally* после блоков *catch* выглядит следующим образом:

```
try {
   // Защищенный код
} catch(ИсключениеТип1 e1) {
   // Блок catch
} catch(ИсключениеТип2 e2) {
   // Блок catch
} catch(ИсключениеТип3 e3) {
   // Блок catch
} finally {
   // Блок finally всегда выполняется
}
```

Вне зависимости от того, возникла ли исключительная ситуация в блоке *try*, задан ли подходящий блок *catch*, не возникла ли ошибка в самом блоке *catch*, все равно блок *finally* будет в конце концов исполнен.

Последовательность выполнения такой конструкции следующая: если оператор *try* выполнен нормально, то будет выполнен блок *finally*. В свою очередь, если оператор *finally* выполняется нормально, то и весь оператор *try* выполняется нормально.

Если во время выполнения блока *try* возникает исключение, существует оператор *catch*, который перехватывает данный тип исключения, происходит выполнение блока, связанного с *catch*. Если блок *catch* выполняется нормально, либо ненормально, все равно затем выполняется блок *finally*. Если блок *finally* завершается нормально, то оператор *try* завершается так же, как завершился блок *catch*.

Если в списке операторов *catch* не находится такого, который обработал бы возникшее исключение, то все равно выполняется блок *finally*. В этом случае, если *finally* завершится нормально, весь *try* завершится ненормально по той же причине, по которой было нарушено исполнение *try*.

Во всех случаях, если блок *finally* завершается ненормально, то весь *try* завершится ненормально по той же причине.

Рассмотрим следующий пример:

```
public class Test {
   public static void main(String args[ ]) {
      int array[] = new int[2];
      try {
         System.out.println("Доступ к элементу:" + array[3]);
      } catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Исключение:" + e);
      } finally {
         array[0] = 6;
         System.out.println("Значение первого элемента: " +
array[0]);
         System.out.println("Оператор finally выполнен.");
      }
   }
}
```

Будет получен следующий результат:

```
Исключение:java.lang.ArrayIndexOutOfBoundsException: 3
Значение первого элемента: 6
Оператор finally выполнен.
```

При использовании блока *try/catch/finally* следует помнить следующее:

- Выражение *catch* не может существовать без оператора *try*.
- Выражение *finally* не может существовать без оператора *try*.
- При наличии блока *try/catch*, выражение *finally* не является обязательным.
- Блок *try* не может существовать при отсутствии выражения *catch* либо выражения *finally*.
- Существование какого-либо кода в промежутке между блоками *try, catch, finally* является невозможным.
- Если срабатывает один из блоков *catch*, то остальные блоки в данной конструкции *try/catch* выполняться не будут.
- Управление никогда не возвращается из блока *catch* обратно в блок *try*; после выполнения блока *catch* управление передается строке, следующей сразу после блока *try/catch*.
- Блоки *try* могут быть вложенными.
- Операторы в блоке *try* должны быть окружены фигурными скобками, даже если это одиночная инструкция.
- Область видимости блока *catch* ограничена ближайшим предшествующим блоком *try*, т. е. блок *catch* не может обрабатывать исключение, выброшенное не своим блоком *try*.

12.2. Иерархия исключений

Все классы исключений являются потомками класса *Throwable*. Если в программе возникнет исключительная ситуация, будет сгенерирован объект класса, соответствующего определенному типу исключения. У класса *Throwable* имеются следующие непосредственные подклассы: *Exception* и *Error*. Исключения типа *Error* относятся к ошибкам, возникающим в виртуальной машине Java, а не в прикладной программе. Исключения, произошедшие от *Error*, не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне

JVM. В качестве примера можно привести *StackOverflowError* (переполнение стека), *OutOfMemoryError* (нехватка памяти).

Ошибки, связанные с работой программы, представлены отдельными подклассами, производными от класса *Exception*. Среди этих исключений следует выделить класс *RuntimeException*. Исключения, порожденные от *RuntimeException*, являются непроверяемыми (*unchecked*), и компилятор не требует обязательной их обработки. Ошибки, произошедшие от *Exception* (не являющиеся наследниками *RuntimeException*), являются проверяемыми (*checked*). Это означает, что во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Иерархия исключений представлена на рисунке 4.

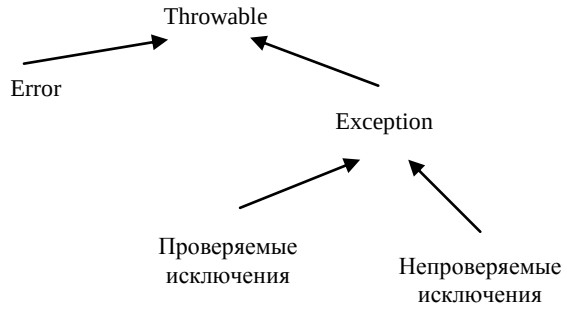

Рисунок 4 – **Иерархия исключений**

Основными методами класса *Throwable* являются следующие:

- *public String getMessage*( ) – возврат подробного сообщения о произошедшем исключении; инициализация данного сообщения производится в конструкторе *Throwable*;
- *public Throwable getCause*( ) – возврат причины исключения, представленной объектом *Throwable*;
- *public String toString*( ) – возврат имени класса, соединенного с результатом *getMessage*( );
- *public void printStackTrace*( ) – выведение результата *toString*( ) совместно с трассировкой стека в *System.err*, поток вывода ошибок;
- *public StackTraceElement*[ ] *getStackTrace*( ) – возврат массива, содержащего каждый элемент в трассировке стека; элемент с номером 0 представляет вершину стека вызовов, последний элемент массива отображает метод на дне стека вызовов;

• *public Throwable fillInStackTrace*( ) заполняет трассировку стека данного объекта *Throwable* текущей трассировкой стека, дополняя какую-либо предшествующую информацию в трассировке стека.

Если в конструкции обработки исключений используется несколько операторов *catch*, классы исключений нужно перечислять в них последовательно, от менее общих к более общим. Перехват исключений из подклассов должен следовать до перехвата исключений из суперклассов. Оператор *catch*, в котором перехватывается исключение из суперкласса, будет перехватывать все исключения из этого суперкласса, а также все исключения из его подклассов. Это означает, что исключения из подкласса вообще не будут обработаны, если попытаться перехватить их после исключений из его суперкласса. Кроме того, недостижимый код считается в Java ошибкой.

Рассмотрим следующий пример:

```
try {
   ...
}
catch(Exception e) {
   ...
}
catch(IOException ioe) {
   ...
}
catch(UserException ue) {
   ...
}
```

В данном примере при возникновении исключительной ситуации (класс, произошедший от *Exception*) будет выполняться всегда только первый блок *catch*. Остальные не будут выполнены ни при каких условиях. Эта ситуация отслеживается компилятором, который сообщает об *UnreachableCodeException* (ошибка – недостижимый код). Правильно будет, если блок *catch*(*Exception e*) в этой конструкции станет последним.

Если метод способен вызвать исключение, порожденное от *Exception*, но не от *RuntimeException*, то должно выполняться одно из следующих условий:

• исключение должно быть обработано в методе с помощью блока *try/catch*;

• в заголовке метода должна стоять конструкция *throw ИсключениеТип*.

Таким образом, если метод не может осуществить обработку контролируемого исключения, производится соответствующее уведомление при использовании ключевого слова *throws*. Ключевое слово *throws* появляется в конце сигнатуры метода, например, *public String readLine*( ) *throws IOException*. Метод также может объявить о том, что им генерируется более чем одно исключение, в случае чего исключения представляются в виде перечня, отделенные друг от друга запятыми.

Вызов метода, в описании которого стоит *throws*, тоже должен находиться внутри блока *try/catch* либо внутри метода с конструкцией *throws* в заголовке и т. д. вплоть до метода *main*( ).

12.3. Переопределение методов и исключения

При переопределении методов следует помнить, что если переопределяемый метод объявляет список возможных исключений, то переопределяющий метод не может расширять этот список, но может его сужать. Рассмотрим следующий пример:

```
public class BaseClass{
   public void method () throws IOException {
   ...
   }
}

public class LegalOne extends BaseClass {
   public void method () throws IOException {
      ...
   }
}

public class LegalTwo extends BaseClass {
   public void method () {
      ...
   }
}

public class LegalThree extends BaseClass {
```

```
    public void method () throws EOFException, MalformedUR-
LException {
        ...
    }
  }

  public class IllegalOne extends BaseClass {
    public void method () throws IOException, IllegalAcces-
sException {
        ...
    }
  }

  public class IllegalTwo extends BaseClass {
    public void method () {
        ...
        throw new Exception();
    }
  }
```

В данном случае можно сделать следующие выводы:

• определение класса *LegalOne* будет корректным, так как переопределение метода *method*( ) верное (список ошибок не изменился);

• определение класса *LegalTwo* будет корректным, так как переопределение метода *method*( ) верное (новый метод не может выбрасывать ошибок, не расширяет список возможных ошибок старого метода);

• определение класса *LegalThree* будет корректным, так как переопределение метода *method*( ) будет верным (новый метод может создавать исключения, которые являются подклассами исключения, проявленного в старом методе, список сузился);

• определение класса *IllegalOne* будет некорректным, так как переопределение метода *method*( ) неверно (*IllegalAccessException* не является подклассом *IOException*, список расширился);

• определение класса *IllegalTwo* будет некорректным, хотя заголовок *method*( ) объявлен верно (список не расширился), в теле метода бросается исключение, не указанное в *throws*.

12.4. Использование оператора *throw*

Помимо того, что предопределенная исключительная ситуация может быть возбуждена исполняющей системой Java, программист сам может сделать ошибку. Делается это с помощью оператора *throw*. Создание исключительной ситуации в программе выполняется с помощью оператора *throw* с аргументом, значение которого может быть приведено к типу *Throwable*, например:

```
throw ex;
throw new IOException();
```

12.5. Создание пользовательских классов исключений

Имеющиеся в стандартной библиотеке классов Java классы исключений описывают большинство исключительных ситуаций, которые могут возникнуть при выполнении программы, но иногда требуется создать свои собственные классы исключений со своей логикой. Создание собственных классов исключений допускается. Для этого достаточно создать свой класс, унаследовав его от любого наследника *java.lang.Throwable* (или от самого *Throwable*). При этом если надо создать исключение, необязательное к перехвату, то его надо унаследовать от класса *RuntimeException*. Если нужно создать контролируемое исключение, то следует расширить класс *Exception*:

```
class MyException extends Exception {
…
}
```

Рассмотрим следующий пример создания и использования собственного исключения:

```
public class InsufficientFundsException extends Exception {
   private double amount;

   public InsufficientFundsException(double amount) {
      this.amount = amount;
   }

   public double getAmount() {
```

```
        return amount;
      }
   }
```

Класс *Checking* содержит метод *withdraw*( ), генерирующий *InsufficientFundsException*. Текст программы имеет следующий вид:

```
   public class Checking {
      private int number;
      private double balance;

      public Checking(int number) {
         this.number = number;
      }

      public void deposit(double amount) {
         balance += amount;
      }

      public void withdraw(double amount) throws InsufficientFundsException {
         if(amount <= balance) {
            balance -= amount;
         }else {
            double needs = amount - balance;
            throw new InsufficientFundsException(needs);
         }
      }

      public double getBalance() {
         return balance;
      }

      public int getNumber() {
         return number;
      }
   }
```

Класс *Bank* демонстрирует вызов методов *deposit*( ) и *withdraw*( ) класса *Checking*. Текст программы имеет следующий вид:

```
public class Bank {

   public static void main(String[ ] args) {
      Checking c = new Checking(101);
      System.out.println("Депозит $300...");
      c.deposit(300.00);

      try {
         System.out.println("\nСнятие $100...");
         c.withdraw(100.00);
         System.out.println("\nСнятие $400...");
         c.withdraw(400.00);
      }catch(InsufficientFundsException e) {
         System.out.println("Извините, но у Вас $" +
e.getAmount());
         e.printStackTrace();
      }
   }
}
```

В результате работы программы будет получен следующий результат:

```
Депозит $300...

Снятие $100...

Снятие $400...
Извините, но у Вас $200.0
InsufficientFundsException
   at Checking.withdraw(Checking.java:25)
   at Bank.main(Bank.java:13)
```

Задания для самостоятельной работы

Задание 12.1. В папке *theme12* создайте папку (package) с именем *exercise12_1*.

В папке *exercise12_1* создайте класс *MyEx*. Тело этого класса будет пустым.

В папке *exercise12_1* создайте класс *ExTestDrive*. Тело этого класса заполните в правильном порядке строками кода, приведенными на рисунке 5. Добавьте недостающие фигурные скобки. Класс *ExTestDrive* содержит метод *main*, первой строчкой которого инициализируется строковая переменная *test*. Если переменная *test* принимает значение *yes*, то программа выведет на консоль *thaws*. Если переменная *test* принимает значение *no*, то программа выведет на консоль *throws*.

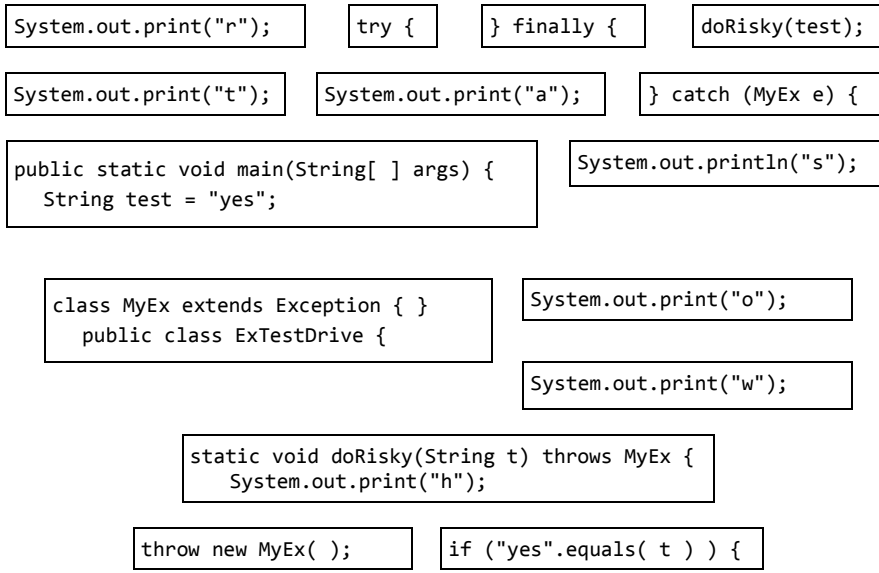

Рисунок 5 – **Части программы к заданию 12.1**

Задание 12.2. В папке *theme12* создайте папку (package) с именем *exercise12_2*.

В папке *exercise12_2* создайте открытый класс-исключение *InsufficientFundsException*.

В классе *InsufficientFundsException* создайте поле *amount* (сумма) типа *double*. Подумайте, какой должен быть модификатор доступа у этого поля. Создайте геттер для поля.

В классе *InsufficientFundsException* создайте конструктор с одним параметром для инициализации поля класса.

В папке *exercise12_2* создайте открытый класс *Checking*.

В классе *Checking* создайте поля number (номер счета) типа *int* и *balance* (баланс) типа *double*. Подумайте, какой должен быть модификатор доступа у этих полей. Создайте геттеры для полей.

В классе *Checking* создайте конструктор с одним параметром для инициализации поля *number*.

В классе *Checking* создайте открытый метод *deposit*( ) (депонировать) с одним параметром *amount* типа *double*. Метод увеличивает баланс на величину *amount*. Метод не возвращает значения.

В классе *Checking* создайте открытый метод *withdraw*( ) (снимать деньги со счета) с одним параметром *amount* типа *double*. Метод не возвращает значения. Если сумма *amount* не превосходит текущего баланса счета *balance*, то уменьшите значение баланса на эту сумму. В противном случае вычислите недостающую сумму и выбросите исключение *InsufficientFundsException* с этой суммой. Метод не обрабатывает никакие исключения.

В папке *exercise12_2* создайте открытый класс *Bank*.

В классе *Bank* создайте метод *main*( ).

В методе *main*( ) создайте экземпляр класса *Checking* с произвольным числом; выведите на консоль фразу «Депозит 300 USD»; вызовите метод *deposit*( ); в блоке *try* выведите на консоль «Снятие 100 USD» и вызовите метод *withdraw*( ), а затем выведите на консоль «Снятие 400 USD» и еще раз вызовите метод *withdraw*( ); в блоке *catch* укажите исключение необходимого типа; в блоке должно быть выведено на консоль «Невозможно выполнить операцию. Не хватает … USD», используйте подходящий метод класса *InsufficientFundsException*.

Задание 12.3. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *ScaryException* (страшное исключение). Тело этого класса пустое.

В папке *exercise12_3* создайте открытый класс *TestException*.

В классе *TestException* создайте закрытый метод *doRisky*( ) с одним параметром-строкой. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Начало опасного метода», затем если параметр имеет значение *yes*, то выбросите исключение *ScaryException*. В конце метода выведите на консоль «Конец опасного метода». Метод не обрабатывает никаких исключений.

В классе *TestException* создайте метод *main*( ). В методе *main*( ) строку *test* инициализируйте некоторым значением, например, «Нет». В блоке *try* выведите на консоль «Начало *try*», затем вызовите метод

doRisky( ) с аргументом *test* и выведите на консоль «Конец *try*». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Жуткое исключение». В блоке *finally* выведите на консоль «Блок *finally*». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.4. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *ClothingException*. Тело этого класса пустое.

В папке *exercise12_3* создайте открытый класс *Washer* (стиральная машина).

В классе *Washer* создайте закрытый метод *doLaundry*( ) (стирать) с одним параметром-строкой. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Идет стирка одежды», затем если параметр имеет значение «Постирано плохо», то выбросите исключение *ClothingException*. В конце метода выведите на консоль «Конец стирки одежды». Метод *doLaundry*( ) не обрабатывает никаких исключений.

В классе *Washer* создайте метод *main*( ). В методе *main*( ) строку *result* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Нужно постирать грязные вещи», затем вызовите метод *doLaundry*( ) с аргументом *result* и выведите на консоль «Конец *try*». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Загрязнения на одежде не отстирались». В блоке *finally* выведите на консоль «Развешиваем сушиться мокрую одежду». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.5. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *BakingException*. Тело этого класса пустое.

В папке *exercise12_3* создайте открытый класс *Cook* (повар).

В классе *Cook* создайте закрытый метод *turnOvenOn*( ). В методе выведите на консоль «Включаю плиту».

В классе *Cook* создайте закрытый метод *turnOvenOff*( ). В методе выведите на консоль «Выключаю плиту».

В классе *Cook* создайте закрытый метод *bake*( ) (приготовить) с одним параметром-строкой. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Готовлю блюдо», затем если параметр имеет значение «Блюдо испортилось», то выбросите исключение *BakingException*. В конце метода выведите на консоль «Конец приготовления блюда». Метод *bake*( ) не обрабатывает никаких исключений.

В классе *Cook* создайте метод *main*( ). В методе *main*( ) строку *result* инициализируйте некоторым значением. В блоке *try* вызовите метод *turnOvenOn*( ), затем вызовите метод *bake*( ) с аргументом *result*. В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Блюдо полностью испортилось». В блоке *finally* вызовите метод *turnOvenOff*( ). В конце метода *main*( ) выведите на консоль «Конец работы повара».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.6. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *HungryPetException*. Тело этого класса пустое.

В папке *exercise12_3* создайте открытый класс *PetOwner* (владелец домашнего животного).

В классе *PetOwner* создайте закрытый метод *feedPet*( ) (кормить домашнего питомца) с одним параметром-строкой. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Кормлю домашнего питомца», затем если параметр имеет значение «Hungry», то выбросите исключение *HungryPetException*. В конце метода выведите на консоль «Накормил!». Метод *feedPet*( ) не обрабатывает никаких исключений.

В классе *PetOwner* создайте метод *main*( ). В методе *main*( ) строку *hunger* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Начало *try*», затем вызовите метод *feedPet*( ) с аргументом *hunger* и выведите на консоль «Конец *try*». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Мало корма». В блоке *finally* выведите на консоль «Блок *finally*». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.7. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooManyItemsException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Мы не сможем доставить столько единиц товара за один раз».

В папке *exercise12_3* создайте открытый класс *Order* (заказ).

В классе *Order* создайте приватную финализированную переменную *maxQuantity* (максимальное количество) типа *byte*.

В классе *Order* создайте конструктор с одним параметром для инициализации переменной класса.

В классе *Order* создайте закрытый метод *checkOrder*( ) (проверить заказ) с одним параметром *quantity* типа *byte*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Проверяем заказ клиента», затем если значение параметра больше *maxQuantity*, то выбросите исключение *TooManyItemsException*. В конце метода выведите на консоль «Заказ проверен». Метод *checkOrder*( ) не обрабатывает никаких исключений.

В классе *Order* создайте метод *main*( ). В методе *main*( ) переменную *selectedQuantity* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Клиенту предлагается сделать заказ», затем вызовите метод *checkOrder*( ) с аргументом *selectedQuantity* и выведите на консоль «Заказ будет доставлен через 5 дней». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Мы не сможем доставить столько единиц товара за один раз», используя подходящий метод. В блоке *finally* выведите на консоль «Блок *finally*». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.8. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *HungryCatException*. Тело этого класса пустое.

В папке *exercise12_3* создайте открытый класс *CatOwner* (владелец домашнего животного).

В классе *CatOwner* создайте закрытый метод *feedCat*( ) (кормить домашнего питомца) с одним параметром-строкой. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Кормлю своего кота», затем если параметр имеет значение «Hungry», то выбросите исключение *HungryCatException*. В конце метода выведите на консоль «Накормил!». Метод *feedCat*( ) не обрабатывает никаких исключений.

В классе *CatOwner* создайте метод *main*( ). В методе *main*( ) строку *hunger* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Начало *try*», затем вызовите метод *feedCat*( ) с аргументом *hunger* и выведите на консоль «Конец *try*». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Мало корма». В блоке *finally* выведите на консоль «Блок *finally*». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.9. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooManyCarsException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Вся парковка занята. Свободных мест нет».

В папке *exercise12_3* создайте открытый класс *Parking* (парковка).

В классе *Parking* создайте приватную финализированную переменную *maxQuantity* (максимальное количество) типа *byte*.

В классе *Parking* создайте конструктор с одним параметром для инициализации переменной класса.

В классе *Parking* создайте закрытый метод *check*( ) (проверить) с одним параметром *quantity* типа *byte*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Ищем место для парковки», затем если значение параметра больше *maxQuantity*, то выбросите исключение *TooManyCarsException*. В конце метода выведите на консоль «Нашли свободное место для парковки». Метод *check*( ) не обрабатывает никаких исключений.

В классе *Parking* создайте метод *main*( ). В методе *main*( ) переменную *carsQuantity* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Приехали на парковку», затем вызовите ме-

тод *check*( ) с аргументом *carsQuantity* и выведите на консоль «Припарковались». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Вся парковка занята. Свободных мест нет», используя подходящий метод. Затем выведите на консоль «Припарковались в другом месте». В блоке *finally* выведите на консоль «Выходим из машины». В конце метода *main*( ) выведите на консоль «Конец поездки».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.10. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooHeavyBagException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Слишком тяжелая сумка».

В папке *exercise12_3* создайте открытый класс *Bag* (сумка).

В классе *Bag* создайте приватную финализированную переменную *maxWeight* (максимальный вес) типа *float*.

В классе *Bag* создайте конструктор с одним параметром для инициализации переменной класса.

В классе *Bag* создайте закрытый метод *checkBag*( ) (проверить сумку) с одним параметром *weight* типа *float*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Проверим, не слишком ли тяжелая сумка», затем если значение параметра больше *maxWeight*, то выбросите исключение *TooHeavyBagException*. В конце метода выведите на консоль «Сумка не тяжелая». Метод *checkBag*( ) не обрабатывает никаких исключений.

В классе *Bag* создайте метод *main*( ). В методе *main*( ) переменную *bagWeight* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Сложили вещи в сумку», затем вызовите метод *checkBag*( ) с аргументом *bagWeight* и выведите на консоль «Берем сумку и едем на вокзал». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Слишком тяжелая сумка», используя подходящий метод. Затем выведите на консоль «Я не могу ее поднять. Придется какие-то вещи выложить». В блоке *finally* выведите на консоль «Надо торопиться, чтобы не опоздать на поезд». В конце метода *main*( ) выведите на консоль «Вот мы и на вокзале».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.11. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooBoringLectureException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Какая скучная лекция!».

В папке *exercise12_3* создайте открытый класс *Lecture* (лекция).

В классе *Lecture* создайте закрытый метод *check*( ) (проверить) с одним параметром типа *boolean*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Идти или не идти на лекцию – вот в чем вопрос», затем если параметр принимает значение *true*, то выбросите исключение *TooBoringLectureException*. В конце метода выведите на консоль «Иду в аудиторию». Метод *check*( ) не обрабатывает никаких исключений.

В классе *Lecture* создайте метод *main*( ). В методе *main*( ) переменную *result* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Прозвенел звонок на лекцию», затем вызовите метод *check*( ) с аргументом *result* и выведите на консоль «Лекция началась». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Какая скучная лекция!», используя подходящий метод. В блоке *finally* выведите на консоль «Лекция закончилась». В конце метода *main*( ) выведите на консоль «Началась перемена».

Задание 12.12. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooBoringBookException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Какая скучная книга!».

В папке *exercise12_3* создайте открытый класс *Book* (лекция).

В классе *Book* создайте закрытый метод *readBook*( ) с одним параметром типа *boolean*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Начинаю читать книгу», затем если параметр принимает значение *true*, то выбросите исключение *TooBoringBookException*. В конце метода выведите на консоль «Про-

читал книгу». Метод *readBook*( ) не обрабатывает никаких исключений.

В классе *Book* создайте метод *main*( ). В методе *main*( ) переменную *result* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Взял в библиотеке книгу почитать», затем вызовите метод *readBook*( ) с аргументом *result* и выведите на консоль «Книга понравилась». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Какая скучная книга!», используя подходящий метод. В блоке *finally* выведите на консоль «Надо сдать книгу в библиотеку». В конце метода *main*( ) выведите на консоль «Надо будет взять еще что-нибудь почитать».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.13. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *HeavyLuggageException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Превышение веса багажа».

В папке *exercise12_3* создайте открытый класс *Luggage* (багаж).

В классе *Luggage* создайте приватную финализированную переменную *maxWeight* (максимальный вес) типа *int*.

В классе *Luggage* создайте конструктор с одним параметром для инициализации переменной класса.

В классе *Luggage* создайте закрытый метод *checkLuggage*( ) (проверить багаж) с одним параметром *weight* типа *float*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Взвесим багаж», затем если значение параметра больше *maxWeight*, то выбросите исключение *HeavyLuggageException*. В конце метода выведите на консоль «Багаж допустимого веса». Метод *checkLuggage*( ) не обрабатывает никаких исключений.

В классе *Luggage* создайте метод *main*( ). В методе *main*( ) переменную *luggageWeight* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Надо пройти регистрацию на рейс», затем вызовите метод *checkLuggage*( ) с аргументом *luggageWeight* и выведите на консоль «Ничего доплачивать не нужно за багаж». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Превышение веса багажа», используя подходящий метод. Затем выведите на консоль «Нужно доплатить по

тарифу за перевозку багажа». В блоке *finally* выведите на консоль «Регистрация пройдена». В конце метода *main*( ) выведите на консоль «Теперь нужно пройти паспортный и таможенный контроль».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.14. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте открытый класс-исключение *TooManyBikesException*. В этом классе создайте конструктор без параметров, в котором вызовите конструктор суперкласса со строковым аргументом «Вся парковка занята. Свободных мест нет».

В папке *exercise12_3* создайте открытый класс *BikeParking* (парковка для велосипедов).

В классе *BikeParking* создайте приватную финализированную переменную *maxQuantity* (максимальное количество) типа *byte*.

В классе *BikeParking* создайте конструктор с одним параметром для инициализации переменной класса.

В классе *BikeParking* создайте закрытый метод *check*( ) (проверить) с одним параметром *quantity* типа *byte*. Метод не возвращает никакого результата. В методе сначала выведите на консоль «Ищем место для парковки велосипеда», затем если значение параметра больше *maxQuantity*, то выбросите исключение *TooManyBikesException*. В конце метода выведите на консоль «Нашли свободное место для парковки». Метод *check*( ) не обрабатывает никаких исключений.

В классе *BikeParking* создайте метод *main*( ). В методе *main*( ) переменную *bikesQuantity* инициализируйте некоторым значением. В блоке *try* выведите на консоль «Приехали на парковку», затем вызовите метод *check*( ) с аргументом *bikesQuantity* и выведите на консоль «Оставили велосипед». В блоке *catch* укажите подходящий тип для перехватываемого исключения. Выведите на консоль «Вся парковка занята. Свободных мест нет», используя подходящий метод. Затем выведите на консоль «Но все-таки нашли, где оставить велосипед». В блоке *finally* выведите на консоль «Можно идти по своим делам». В конце метода *main*( ) выведите на консоль «Конец *main*».

Укажите, какие изменения нужно внести в код метода *main*( ), чтобы программа скомпилировалась, если в методе *main*( ) закомментировать блок *catch*.

Задание 12.15. В папке *theme12* создайте папку (package) с именем *exercise12_3*.

В папке *exercise12_3* создайте два открытых класса исключений *PantsException* и *ShirtException*. В каждом классе напишите конструктор для этого класса, который принимает аргумент *String* и хранит его внутри объекта в ссылке *String*.

В папке *exercise12_3* создайте открытый класс *TestException*.

В классе *TestException* создайте два закрытых метода *f*( ) и *g*( ). В методе *g*( ) выбросите исключение *PantsException*. В методе *f*( ) вызовите метод *g*( ), поймайте его исключение и в блоке *catch* выбросите исключение *ShirtException*.

В классе *TestException* создайте метод *main*( ). В методе *main*( ) вызовите метод *f*( ).

Тема 13. ВВОД-ВЫВОД ДАННЫХ

13.1. Потоковая организация системы ввода-вывода Java

При создании приложений всегда возникает необходимость прочитать информацию из какого-либо источника и сохранить результат. Во многих случаях требуется выводить результаты на принтер, в файл, базу данных или передавать по сети. Исходные данные тоже часто приходится загружать из файла, базы данных или из сети.

Для того чтобы отвлечься от особенностей конкретных устройств ввода-вывода, в Java употребляется понятие потока (stream). Поток – это абстракция, которая выдает и получает информацию. Поток – это объект, способный производить или принимать части данных. Поток связан с физическим устройством ввода или вывода. Поток прячет детали того, что случается с данными внутри реального устройства ввода-вывода. Рассмотрим иерархию классов, содержащихся в пакете *java.io*.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода-вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т. е. связывающих программу с областью оперативной памяти. Кроме того, можно создать канал (pipe) обмена информацией между подпроцессами.

Потоковая организация системы ввода-вывода Java представлена на рисунке 6.

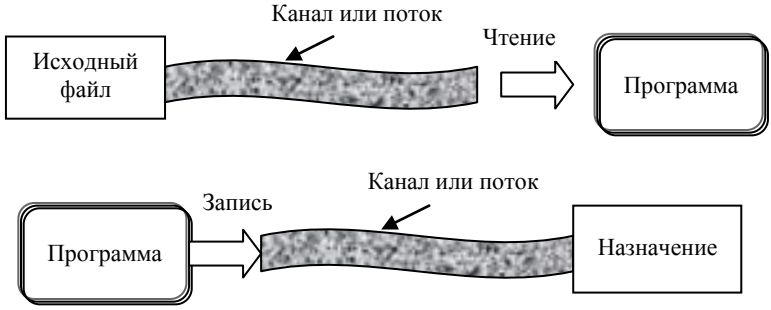

Рисунок 6 – **Потоковая организация системы ввода-вывода Java**

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, – потоком вывода.

В современных версиях Java определены два типа потоков: байтовые и символьные. Первоначально в Java были доступны только байтовые потоки, но вскоре были реализованы и символьные. Байтовые потоки предоставляют удобные средства для управления вводом и выводом байтов. Например, их можно использовать для чтения и записи двоичных данных. Потоки этого типа особенно удобны при работе с файлами. С другой стороны, символьные потоки ориентированы на обмен символьными данными. В них применяется кодировка Unicode, и поэтому их легко интернационализировать. Вместе с тем на самом нижнем уровне все средства ввода-вывода имеют байтовую организацию.

Необходимость поддерживать два разных типа потоков ввода-вывода привела к созданию двух иерархий классов: одна для байтовых, другая – для символьных данных. Байтовые потоки определены с использованием двух иерархий классов, на вершинах которых находятся абстрактные классы *InputStream* и *OutputStream* соответственно. На вершине иерархии классов, поддерживающих символьные потоки, находятся абстрактные классы *Reader* и *Writer*.

13.2. Байтовые потоки ввода

Иерархия классов байтовых потоков ввода представлена на рисунке 7.

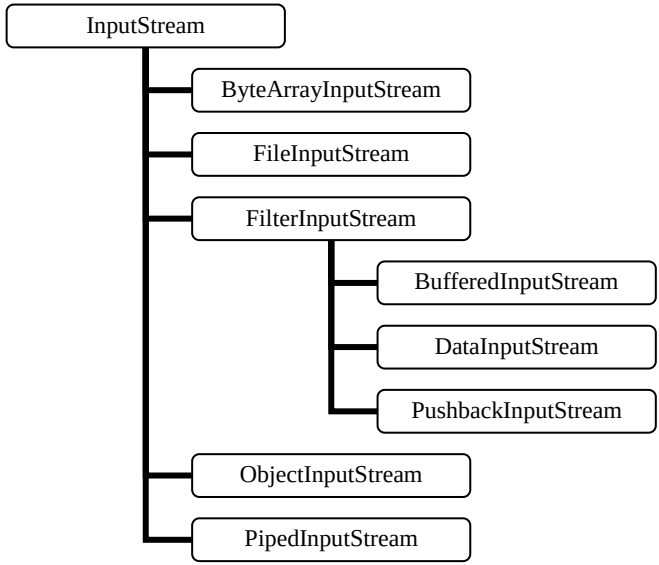

Рисунок 7 – **Иерархия классов байтовых потоков ввода**

Классы-потомки класса *InputStream* являются конкретными подклассами, реализующими различные функциональные возможности и учитывающими особенности обмена данными с разными устройствами (таблица 7).

Таблица 7 – **Классы байтовых потоков ввода**

Класс	Описание
InputStream	Абстрактный класс, описывающий потоковый ввод
ByteArrayInputStream	Входной поток для чтения из байтового массива
FileInputStream	Входной поток для чтения из файла
FilterInputStream	Реализация класса *InputStream*. Это просто базовый класс для трех других классов
BufferedInputStream	Буферизованный поток ввода. Он служит для организации более эффективного «буферизованного» ввода данных
DataInputStream	Поток ввода, содержащий методы для чтения данных стандартных типов, определенных в Java
PushbackInputStream	Поток ввода, поддерживающий возврат одного байта обратно в поток ввода

Окончание таблицы 7

Класс	Описание
ObjectInputStream	Входной поток для объектов
PipedInputStream	Канал ввода. Это специальный класс, используемый для связи отдельных программ друг с другом

Методы, определенные в классе *InputStream*, перечислены в таблице 8.

Таблица 8 – **Методы класса *InputStream***

Метод	Описание метода
int available( )	Возвращает количество байтов ввода, доступных в данный момент для чтения
void close( )	Закрывает источник ввода. Дальнейшие попытки чтения будут генерировать исключение *IOException*
void mark(int readLimit)	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром *readLimit*
boolean markSupported( )	Возвращает значение *true*, если методы *mark*( ) и *reset*( ) поддерживаются вызывающим потоком
abstract int read( )	Возвращает целочисленное представление следующего байта в потоке. При достижении конца потока возвращается значение –1
int read (byte b[ ])	Пытается прочитать *b.length* байтов в массив *b*, возвращая фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение –1
int read(byte[ ] b, int off, int len)	Пытается прочитать *len* байтов в массив *b*, начиная с элемента *b*[*off*], и возвращает фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение –1
void reset( )	Сбрасывает входной указатель на ранее установленную метку
long skip(long n)	Пропускает n входных байтов, возвращая фактическое количество пропущенных байтов

Все методы класса *InputStream* (за исключением методов *markSupported*( ) и *mark*(*int readlimit*)) могут порождать *IOException.*

Для считывания данных из файла предназначен класс *FileInputStream*, который является наследником класса *InputStream* и поэтому реализует все его методы.

Для создания объекта *FileInputStream* можно использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает следующий путь к считываемому файлу:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение *FileNotFoundException*.

Завершив операции с файлом, следует закрыть его с помощью метода *close*( ). При закрытии файла освобождаются связанные с ним системные ресурсы, которые вновь можно будет использовать для работы с другими файлами. Если этого не сделать, возможна утечка памяти из-за того, что часть памяти остается выделенной для ресурсов, которые больше не используются.

Рассмотрим следующий пример, в котором читается содержимое файла с помощью метода *read*( ), а затем оно выводится на консоль:

```
import java.io.*;
class ShowFile (
   public static void main(String args[ ]) {
      int i;
      // Инициализация переменной fin значением null
      FileinputStream fin = null;
      // Открытие файла, чтение из него символов, пока
      // не встретится признак конца файла EOF, и
      // последующее закрытие файла в блоке finally
      try {
         fin = new FileinputStream("D://SomeDir//note.txt");
         do {
            i = fin.read();
            if(i != -1) System.out.print((char) i);
         } while(i != -1);
      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода");
      } finally {
         // Файл закрывается в любом случае
         try {
```

```
            if(fin != null) fin.close(); //Закрыть fin,
если он не null
          } catch(IOException exc) {
            System.out.println("Ошибка при закрытии файла");
          }
        }
      }
   }
```

Считать данные в массив байтов, затем производить с ним манипуляции можно следующим образом:

```
byte[] buffer = new byte[fin.available()];
// считаем файл в буфер
fin.read(buffer, 0, fin.available());
System.out.println("Содержимое файла:");
for(int i = 0; i < buffer.length; i++){
   System.out.print((char) buffer[i]);
}
```

Класс *ByteArrayInputStream* представляет входной поток, использующий в качестве источника данных массив байтов. Он имеет следующие конструкторы:

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
```

В качестве параметров конструкторы используют массив байтов *buf*, из которого производится считывание, смещение относительно начала массива *offset* и количество считываемых символов *length*.

Считаем массив байтов и выведем его на экран следующим образом:

```
import java.io.*;
public class FilesApp {
   public static void main(String[ ] args) {

      byte[] array1 = new byte[]{1, 3, 5, 7};
      ByteArrayInputStream byteStream1 = new ByteArrayIn-
putStream(array1);
      int b;
      while((b=byteStream1.read())!=-1){
```

```
            System.out.println(b);
        }

        String text = "Hello world!";
        byte[] array2 = text.getBytes();
        ByteArrayInputStream byteStream2 = new ByteArrayIn-
putStream(array2, 0, 5);
        int c;
        while((c = byteStream2.read()) != -1){
            System.out.println((char)c);
        }
    }
}
```

В отличие от других классов потоков для закрытия объекта *ByteArrayInputStream* не требуется вызывать метод *close*.

13.3. Байтовые потоки вывода

Иерархия классов байтовых потоков вывода представлена на рисунке 8.

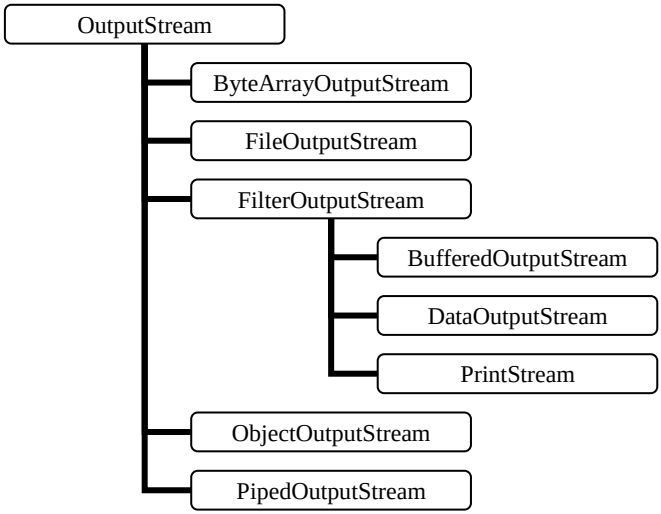

Рисунок 8 – **Иерархия классов байтовых потоков вывода**

Описание классов байтовых потоков вывода приводится в таблице 9.

Таблица 9 – **Классы байтовых потоков вывода**

Класс	Описание
OutputStream	Абстрактный класс, описывающий потоковый вывод
ByteArrayOutputStream	Выходной поток для записи в байтовый массив
FileOutputStream	Выходной поток для записи из файла
FilterOutputStream	Реализация класса *OutputStream*
BufferedOutputStream	Буферизованный выходной поток. Служит для организации более эффективного буферизованного вывода в поток
DataOutputStream	Поток ввода, содержащий методы для записи стандартных типов данных Java
PrintStream	Выходной поток, включающий методы *print*( ) и *println*( )
ObjectOutputStream	Выходной поток для объектов
PipedOutputStream	Канал вывода. Класс предназначен для передачи информации между программами через каналы (*pipes*)

Методы, определенные в классе *OutputStream*, перечислены в таблице 10.

Таблица 10 – **Методы класса *OutputStream***

Метод	Описание
void close( )	Закрывает выходной поток. Дальнейшие попытки записи будут генерировать исключение *IOException*
void flush( )	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
abstract void write(int b)	Записывает один байт в выходной поток
void write (byte b[ ])	Записывает полный массив байтов в выходной поток
void write(byte[ ] b, int off, int len)	Записывает часть массива *b* в количестве *len* байтов, начиная с элемента *b*[*off*]

Класс *FileOutputStream* предназначен для записи байтов в файл. Он является производным от класса *OutputStream*, поэтому наследует всю его функциональность. Имеет следующие конструкторы:

```
   public FileOutputStream(File file) throws FileNotFoundEx-
ception
   public FileOutputStream(String name) throws FileNotFoun-
dException
   public FileOutputStream(String name, boolean append)
   throws FileNotFoundException
```

Все три конструктора принимают в качестве параметра путь к файлу для записи. Если файл не существует, то он создается. Если файл существует, то он будет полностью обновлен. Если открыть и сразу закрыть файл, то реальный файл на диске будет нулевой длины. Исключением из предыдущего правила является последний из конструкторов. Если в нем в качестве третьего параметра (append) указать *true*, то файл будет дописываться.

В следующем примере осуществляется копирование текстового файла. В папке проекта должен быть создан файл *file.txt* с любым или пустым содержимым. Текст программы имеет следующий вид:

```
   import java.io.*;
   public class FileCopy {
      public static void main(String args[ ]) throws IOExcep-
tion {
         FileInputStream fileIn = null;
         FileOutputStream fileOut = null;

         try {
            fileIn = new FileInputStream("file.txt");
            fileOut = new FileOutputStream("copied_file.txt");

            int a;
      // Чтение содержимого файла file.txt и запись в файл
copied_file.txt
            while ((a = fileIn.read()) != -1) {
               fileOut.write(a); //
            }
         } finally {
            if (fileIn != null) fileIn.close();
            if (fileOut != null) fileOut.close();
         }
      }
   }
```

Класс *ByteArrayOutputStream* представляет поток вывода, использующий массив байтов в качестве места вывода. Чтобы создать объект данного класса, можно использовать один из следующих конструкторов:

```
ByteArrayOutputStream()
ByteArrayOutputStream(int size)
```

Первый конструктор создает массив для хранения байтов длиной в 32 байта (при необходимости его размер увеличивается), а второй создает массив длиной *size*.

Рассмотрим применение следующего класса:

```
import java.io.*;
public class FilesApp {

   public static void main(String[ ] args) {
      ByteArrayOutputStream baos = new ByteArrayOutput-
Stream();
      String text = "Hello Wolrd!";
      byte[] buffer = text.getBytes();
      try{
         baos.write(buffer);
      } catch(Exception ex){
         System.out.println(ex.getMessage());
      }
      // превращаем массив байтов в строку
      System.out.println(baos.toString());

      // получаем массив байтов и выводим посимвольно
      byte[] array = baos.toByteArray();
      for(byte b: array){
         System.out.print((char)b);
      }
      System.out.println();
   }
}
```

Для объектов *ByteArrayInputStream* и для *ByteArrayOutputStream* не надо явным образом закрывать поток с помощью метода *close*.

Класс *ByteArrayOutputStream* имеет метод *writeTo*(*OutputStream out*), записывающий содержимое одного потока в другой. Метод принимает в качестве параметра объект *OutputStream*, в который производится запись массива байт:

```
byte[ ] buf = {‘a’, ‘b’, ‘c’, ‘d’};
ByteArrayOutputStream baos = new ByteArrayOutputStream( );
baos.write(buf);
FileOutputStream out = new FileOutputStream(“result.txt”);
baos.writeTo(out); // равносильно методу out.write(buf);
```

В пакете *java.lang* есть класс с именем *System*, в котором определены следующие статические переменные:

```
public static InputStream in;
public static PrintStream out;
public static PrintStream err;
```

Данные переменные являются стандартными потоками ввода, вывода и вывода ошибок соответственно. Эти потоки связаны с консолью, но могут быть переназначены на другое устройство.

13.4. Оператор *try* с ресурсами

В примерах программ, представленных ранее, для закрытия файлов, которые больше не нужны, метод *close*( ) вызывался явным образом. Такой способ закрытия файлов используется еще с тех пор, как вышла первая версия Java. Однако в версию JDK 7 включено новое средство, предоставляющее другой, более рациональный способ управления ресурсами, в том числе и потоками файлового ввода-вывода, автоматизирующий процесс закрытия файлов. Этот способ основывается на новой разновидности оператора *try*, называемой оператором *try* с ресурсами (*try-with-resources*), а иногда – автоматическим управлением ресурсами. Главное преимущество оператора *try* с ресурсами заключается в том, что он предотвращает ситуации, в которых файл (или другой ресурс) непреднамеренно остается неосвобожденным и после того, как необходимость в его использовании отпала. Если не позаботиться о своевременном закрытии файлов, то это может привести к утечке памяти и прочим осложнениям в работе программы.

Общая форма оператора *try* с ресурсами имеет следующий вид:

```
try (описание_ресурса) {
   // использовать ресурс
}
```

Здесь описание ресурса включает в себя объявление и инициализацию ресурса, такого как файл. В это описание входит объявление переменной, которая инициализируется ссылкой на объект управляемого ресурса. По завершении блока *try* объявленный ресурс автоматически освобождается. Если этим ресурсом является файл, то он автоматически закрывается, что избавляет от необходимости вызывать метод *close*( ) явным образом. Оператор *try* с ресурсами также может включать блоки *catch* и *finally*.

Область применимости таких операторов *try* ограничена ресурсами, которые реализуют интерфейс *AutoCloseable*, определенный в пакете *java.lang*. В этом интерфейсе определен метод *close*( ). Интерфейс *AutoCloseable* наследуется интерфейсом *Closeable*, определенным в пакете *java.io*. Оба интерфейса реализуются классами потоков, в том числе *FileInputStream* и *FileOutputStream*. Следовательно, оператор *try* с ресурсами может применяться вместе с потоками, включая потоки файлового ввода-вывода.

С помощью одного подобного оператора *try* можно управлять несколькими ресурсами. Для этого достаточно указать список объявлений ресурсов, разделенных точкой с запятой. В качестве примера ниже приведена переработанная версия программы *FileCopy*:

```
   import java.io.*;
   public class FileCopy {

   public static void main(String args[ ]) throws IOExcep-
tion {
     try   (FileInputStream   fileIn   =   new   FileInput-
Stream("file.txt");
        FileOutputStream fileOut =
           new FileOutputStream("copied_file.txt"))
     {
        int a;
   // Чтение содержимого файла file.txt и запись в файл
copied_file.txt
        while ((a = fileIn.read()) != -1) {
```

```
            fileOut.write(a); //
        }
    }
}
```

13.5. Буферизованный ввод-вывод

Операции ввода-вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область – буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Житейский пример буфера – почтовый ящик, в котором накапливаются письма. В него бросают письмо, уходят по своим делам, не дожидаются приезда почтовой машины. Почтовая машина периодически очищает почтовый ящик, перенося сразу большое число писем. Необходимо представить себе город, в котором нет почтовых ящиков, толпа людей с письмами в руках дожидается приезда почтовой машины.

Буферизованные потоки являются расширением классов фильтруемых потоков, в них к потокам ввода-вывода присоединяется буфер в памяти. Этот буфер выполняет следующие основные функции:

- он дает возможность исполняющей среде Java проделывать за один раз операции ввода-вывода с более чем одним байтом, тем самым повышая производительность среды;
- поскольку у потока есть буфер, становятся возможными операции пропуска данных в потоке, установки меток и очистки буфера.

Буферизация может быть добавлена к любому байтовому или символьному потоку с помощью специальных классов-оболочек.

Байтовые буферизованные потоки – это *BufferedInputStream* и *BufferedOutputStream*. В конструкторы этих классов нужно передать байтовый поток ввода или вывода соответственно. В конструкторе *BufferedInputStream*(*InputStream in*) создается буферизованный байтовый поток ввода. Для него используется буфер длиной 32 байта. Во втором конструкторе *BufferedInputStream*(*InputStream in, int size*) размер буфера для создаваемого потока задается вторым параметром конструктора. В общем случае оптимальный размер буфера зависит от операционной системы, количества доступной оперативной памяти и конфигурации компьютера.

Вывод в объект *BufferedOutputStream* идентичен выводу в любой *OutputStream* с той разницей, что новый подкласс содержит дополнительный метод *flush*, применяемый для принудительной очистки буфера и физического вывода на внешнее устройство хранящейся в нем информации. Первая форма конструктора этого класса *BufferedOutputStream*(*OutputStream out*) создает поток с буфером размером 32 байта. Вторая форма *BufferedOutputStream*(*OutputStream out*, *int size*) позволяет задавать требуемый размер буфера.

13.6. Чтение и запись двоичных данных

Классы *DataOutputStream* и *DataInputStream* поддерживают двоичный ввод-вывод простых типов данных (*boolean*, *char*, *byte*, *short*, *int*, *long*, *float* и *double*) и строк типа *String*.

Ниже приведен конструктор класса *DataOutputStream*. При вызове ему передается экземпляр класса *OutputStream*:

```
DataOutputStream(OutputStream outputStream)
```

Здесь *outputStream* – выходной поток, в который записываются данные. Например, для того чтобы организовать запись данных в файл, следует передать конструктору в качестве параметра *outputStream* объект типа *FileOutputStream*.

Наиболее число применяемые методы класса *DataOutputStream* приведены в таблице 11. Каждый их них может генерировать исключение *IOException*.

Таблица 11 – **Методы класса *DataOutputStream***

Метод	Описание
void writeBoolean(boolean v)	Записывает в поток логическое однобайтовое значение
void writeByte(int v)	Записывает в поток однобайтовое значение *byte*
void writeShort(int v)	Записывает в поток двухбайтовое значение *short*
void writeChar(int v)	Записывает двухбайтовое значение *char*
void writeInt(int v)	Записывает в поток четырехбайтовое целочисленное значение *int*
void writeLong(long v)	Записывает в поток восьмибайтовое значение *long*
void writeFloat(float v)	Записывает в поток четырехбайтовое значение *float*

Окончание таблицы 11

Метод	Описание
void writeDouble(double v)	Записывает в поток восьмибайтовое значение *double*
void writeBytes(String s)	Записывает в поток строку *s* как последовательность байтов
void writeChars(String s)	Записывает в поток строку *s* как последовательность символов
void writeUTF(String s)	Записывает в поток строку *s* в кодировке UTF-8

Конструктор класса *DataInputStream* имеет следующий вид:

```
DataInputStream(InputStream inputStream)
```

InputStream – это поток, связанный с создаваемым экземпляром класса *DataInputStream*. Для того чтобы организовать чтение данных из файла, следует передать конструктору в качестве параметра *inputStream* объект типа *FileInputStream.*

Принципы работы потоков *DataOutputStream* и *DataInputStream* показаны на рисунке 9.

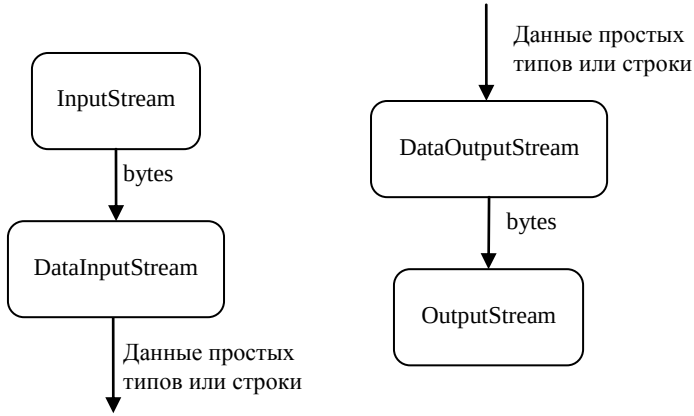

Рисунок 9 – **Чтение и запись двоичных данных**

Наибольшее число применяемых методов класса *DataInputStream* приведено в таблице 12. Каждый их них может генерировать исключение *IOException*.

Таблица 12 – **Методы класса *DataInputStream***

Метод	Описание
boolean readBoolean( )	Читает значение типа *boolean*
byte readByte( )	Читает значение типа *byte*
short readShort( )	Читает значение типа *short*
char readChar( )	Читает значение типа *char*
int readInt( )	Читает значение типа *int*
long readLong( )	Читает значение типа *long*
float readFloat( )	Читает значение типа *float*
double readDouble( )	Читает значение типа *double*
String readUTF( )	Возвращает строку в кодировке *Unicode*

Ниже приведен пример программы, демонстрирующий использование классов *DataOutputStream* и *DataInputStream*. В этой программе данные разных типов сначала записываются в файл, а затем читаются из него.

Текст программы имеет следующий вид:

```
import java.io.*;

public class RWdataDemo {
   public static void main(String[ ] args) {
      int i = 10;
      double d = 1023.56;
      boolean b = true;
      String str = "My short string";

      try (DataOutputStream out =
         new     DataOutputStream(new     FileOutputStream
("datatest.txt")))
      {
         System.out.println("Записано: " + i);
         out.writeInt(i);

         System.out.println("Записано: " + d);
         out.writeDouble(d);
```

```
            System.out.println("Записано: " + b);
            out.writeBoolean(b);

            System.out.println("" + str);
            out.writeUTF(str);
         } catch (IOException e) {
            System.out.println("Ошибка при записи");
         }

         try (DataInputStream in =
            new      DataInputStream(new      FileInputStream
("datatest.txt")))
         {
            System.out.println(in.readInt());
            System.out.println(in.readDouble());
            System.out.println(in.readBoolean());
            System.out.println(in.readUTF());
         } catch (IOException e) {
            System.out.println("Ошибка при чтении");
         }
      }
   }
```

13.7. Символьные потоки

В Java символы хранятся в кодировке Unicode. Символьный поток ввода-вывода автоматически транслирует символы между форматом Unicode и локальной кодировкой пользовательского компьютера.

На вершине иерархии классов, поддерживающих символьные потоки, находятся абстрактные классы *Reader* и *Writer* (рисунок 10). Подклассы абстрактных классов *Reader* и *Writer* применяются для обработки символьных потоков в формате Unicode и почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации. Например, аналогом класса *FileInputStream* является класс *FileReader*. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае.

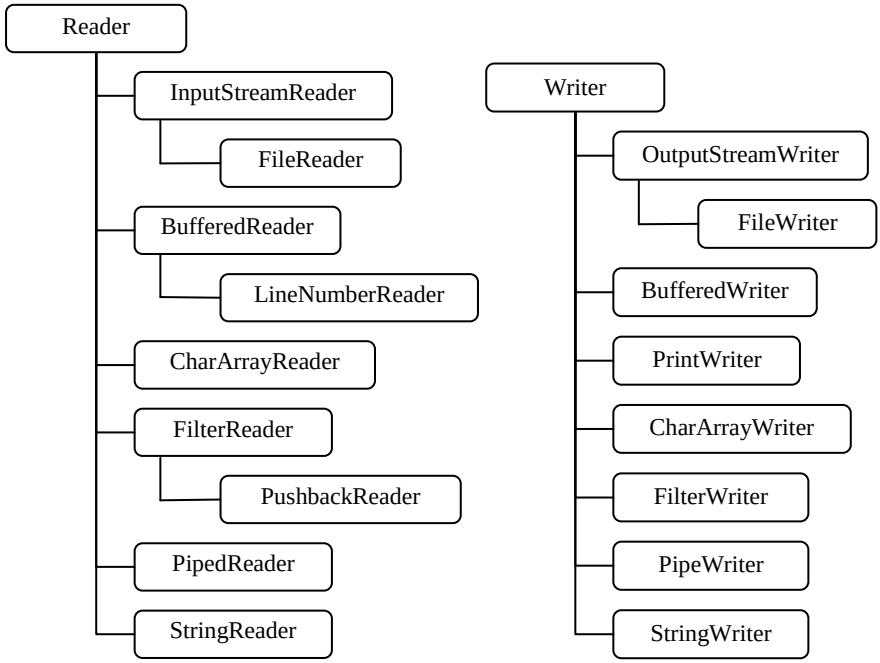

Рисунок 10 – **Классы символьных потоков**

В таблицах 13, 14 приведены методы классов *Reader* и *Writer*. Большинство этих методов может генерировать исключение *IOException*.

Таблица 13 – **Методы, определенные в классе *Reader***

Метод	Описание
abstract void close( )	Закрывает источник ввода
void mark(int numChars)	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром *numChars*
boolean markSupported( )	Возвращает значение *true*, если методы *mark*( ) и *reset*( ) поддерживаются вызывающим потоком
int read( )	Возвращает целочисленное представление следующего символа в вызывающем входном потоке. По достижении конца потока возвращается значение –1

Окончание таблицы 13

Метод	Описание
int read(char buffer[ ])	Пытается прочитать *buffer.length* символов в массив *buffer*, возвращая фактическое количество успешно прочитанных символов. По достижении конца потока возвращается значение –1
abstract int read(char buffer[ ], int offset, int numChars)	Пытается прочитать количество символов, определяемое параметром *numChars*, в массив *buffer*, начиная с элемента *buffer*[*offset*]. По достижении конца потока возвращается значение –1
int read(CharBuffer buffer)	Пытается заполнить буфер, определяемый параметром *buffer*, возвращает количество успешно прочитанных символов. По достижении конца потока возвращается значение –1. *CharBuffer* – это класс, инкапсулирующий последовательность символов, например строку
boolean ready( )	Возвращает значение *true*, если следующий запрос на получение символа может быть выполнен без ожидания. В противном случае возвращается значение *false*
void reset( )	Сбрасывает входной указатель на ранее установленную метку
long skip(long numChars)	Пропускает *numChars* символов во входном потоке, возвращая фактическое количество пропущенных символов

Таблица 14 – **Методы, определенные в классе *Writer***

Метод	Описание
Writer append(char ch)	Добавляет символ *ch* в конец вызывающего выходного потока, возвращая ссылку на вызывающий поток
Writer append(CharSequencechars)	Добавляет последовательность символов *chars* в конец вызывающего потока, возвращая ссылку на вызывающий поток. *CharSequence* – это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме «только чтение»
Writer append(CharSequencechars, int begin, int end)	Добавляет последовательность символов *chars* в конец текущего потока, начиная с позиции, определяемой параметром *begin*, и заканчивая позицией, определяемой параметром *end*. Возвращает ссылку на вызывающий поток. *CharSequence* – это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме «только чтение»

Окончание таблицы 14

Метод	Описание
abstract void close( )	Закрывает выходной поток. Дальнейшие попытки чтения будут генерировать исключение *IOException*
abstract void flush( )	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
void write(int ch)	Записывает один символ в вызывающий выходной поток. Обратите внимание на то, что параметр имеет тип *int*, что позволяет вызывать метод *write*( ) с выражениями, не приводя их к типу *char*
void write(char buffer[ ])	Записывает полный массив символов *buffer* в вызывающий выходной поток
abstract void write (char buffer[ ], int offset, int numChars)	Записывает часть массива символов *buffer* в количестве *numChars* символов, начиная с элемента *buffer*[*offset*], в вызывающий выходной поток
void write(String str)	Записывает строку *str* в вызывающий выходной поток
void write (String str, int offset, int numChars)	Записывает часть строки *str* в количестве *numChars* символов, начиная с позиции, определяемой параметром *offset*, в вызывающий поток

Для байтовых и символьных потоков существуют следующие буферизованные потоки: *BufferedReader* и *BufferedWriter*. Класс *BufferedReader* имеет следующий важный метод:

```
public String readLine() throws IOException
```

Этот метод позволяет читать строку из входного потока. В комбинации с другими классами Java этот метод позволяет организовать ввод с разбиением на слова, вводить числа и т. д. При достижении конца файла метод *readLine*( ) возвращает ссылку *null*.

Основным конструктором класса *BufferedReader* является следующий:

```
public BufferedReader(Reader in)
```

Отдельно этот класс использовать нельзя, только в комбинации с другим классом, например с *FileReader*. Класс *BufferedReader* ранее был использован при чтении данных с консоли. На рисунке 11 представлены потоки ввода при чтении с клавиатуры.

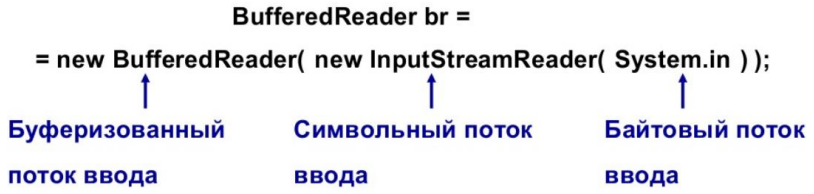

Рисунок 11 – **Потоки ввода при чтении с клавиатуры**

13.8. Использование класса *FileWriter*

Наиболее часто используются следующие конструкторы класса:

```
FileWriter(String имя_файла) throws IOException
FileWriter(String имя_файла, boolean append) throws IOEx-
ception
```

Здесь *имя_файла* обозначает полный путь к файлу. Если параметр *append* принимает значение *true*, данные записываются в конец файла, в противном случае запись осуществляется поверх существующих данных. При возникновении ошибки в каждом из указанных конструкторов генерируется исключение *IOException*.

Работу с классом *FileWriter* можно продемонстрировать на примере небольшой программы, в которой текстовые строки вводятся с клавиатуры, а потом записываются в файл *test.txt*. Набираемый текст читается до тех пор, пока пользователь не введет слово *stop*.

Текст программы имеет следующий вид:

```
import java.io.*;

class Example {
   public static void main(String args[ ]) {
      String str;
      BufferedReader br =
                      new  BufferedReader(new  InputStrea-
mReader(System.in));
      System.out.println("Признак  конца  ввода  -  строка
'stop' ");
```

```
        // Создание объекта FileWriter
        try (FileWriter fw = new FileWriter("test.txt")) {
           do {
              str = br.readLine();
              if(str.compareTo("stop") == 0) break;

              str = str + "\r\n"; // добавить символы перево-
да строки
              fw.write(str); //Запись текстовых строк в файл
           } while(str.compareTo("stop") != 0);
        } catch(IOException exc) {
           System.out.println("Ошибка ввода-вывода: " + exc);
        }
     }
  }
```

13.9. Использование класса *FileReader*

Чаще всего используется следующая форма конструктора этого класса:

```
FileReader(String имя_файла) throws FileNotFoundException
```

Имя_файла обозначает полный путь к файлу. Если указанный файл не существует, генерируется исключение *FileNotFoundException*.

Работу с классом *FileReader* можно продемонстрировать на примере программы для чтения данных из файла *test.txt* и вывода их на экран.

Текст программы имеет следующий вид:

```
import java.io.*;

class Example {
   public static void main(String args[ ]) {
      String s;
      // Создать и использовать объект FileReader, поме-
щенный
      // в оболочку на основе класса BufferedReader
      try (BufferedReader br =
         new BufferedReader(new FileReader("test.txt"))) {
```

```
        // Чтение строк из файла и отображение их на экране
           while((s = br.readLine()) != null) {
              System.out.println(s);
           }
        } catch(IOException exc) {
           System.out.println("Ошибка ввода-вывода: " + exc);
        }
     }
  }
```

Для потока *FileReader* создается оболочка на основе класса *BufferedReader*. Благодаря этому появляется возможность обращаться к методу *readLine*( ). Закрытие потока типа *BufferedReader*, на который в данном примере ссылается переменная *br*, автоматически приводит к закрытию файла.

13.10. Сериализация объектов

Кроме данных базовых типов в поток можно отправлять объекты классов.

Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией.

Существуют следующие способы сделать объект сериализуемым: с использованием интерфейса *java.io.Serializable* или интерфейса *java.io.Externalizable*. Далее рассмотрим только первый интерфейс.

Класс (или один из его суперклассов) должен реализовывать интерфейс *java.io.Serializable* для сериализации объектов этого класса. Интерфейс *Serializable* не имеет методов и только маркирует класс, чтобы можно было идентифицировать его как сериализуемый. Только поля объекта сериализованного класса могут быть сохранены. Методы или конструкторы не сохраняются как части сериализованного потока.

Если какой-либо объект действует как ссылка на другой объект, то поля этого объекта также сериализованы, если класс этого объекта реализует интерфейс *Serializable*. Другими словам, получаемый таким образом граф этого объекта сериализируется полностью. Граф объекта включает дерево или структуру полей объекта и его подобъектов.

Следующим шагом является сохранение объекта. Оно выполняется при помощи класса *java.io.ObjectOutputStream*. Для записи объек-

тов в поток достаточно вызвать метод *writeObject*(*Object ob*) этого класса для сериализации объекта *ob* и пересылки его в выходной поток данных, например:

```
   FileOutputStream fileStream = new FileOutputStream
("MyGame.ser");
   ObjectOutputStream os = new ObjectOutputStream(file
Stream);
   os.writeObject(characterOne);
   os.writeObject(characterTwo);
```

Поля класса, помеченные как *static* или *transient*, будут пропущены при сериализации. Спецификаторы *transient* и *static* означают, что поля, помеченные ими, не могут быть предметом сериализации, но существует различие в десериализации. Поле со спецификатором *transient* после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением *null*), а поле со спецификатором *static* получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта.

Десериализация во многом похожа на сериализацию в обратном порядке:

```
   FileInputStream fileStream = new FileInputStream ("My-
Game.ser");
   ObjectInputStream os = new ObjectInputStream(fileStream);
   GameCharacter elf = (GameCharacter) os.readObject();
```

Рассмотрим процесс десериализации подробно. Например, метод *writeObject* вызывается, чтобы записать единственный объект, названный *a*. Этот объект содержит ссылки на объекты *b* и *c*, объект *b* содержит ссылки на *d* и *e* (рисунок 12). Вызов метода *writeobject*(*a*) записывает не только *a*, но и все объекты, необходимые, чтобы затем при десериализации восставновить все ссылки объекта *a*. Все четыре объекта также будут записаны в поток. Когда затем объект *a* читается из потока методом *readObject*, другие четыре объекта также читаются из потока, все исходные ссылки на объект сохраняются (рисунок 13).

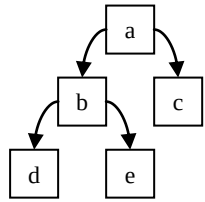

Рисунок 12 – **Граф ссылок объекта *a***

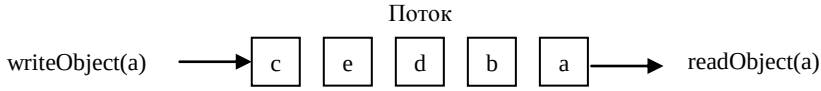

Рисунок 13 – **Запись в поток и чтение из него объекта**

Задания для самостоятельной работы

Задание 13.1. В программе с консоли введите два имени файла: *файл1* и *файл2*. *Файл1* копируется на место, заданное вторым файлом с именем *файл2*. Если файла с именем *файл1* не существует, то программа должна вывести надпись «Файл не существует.», еще раз прочитать имя файла с консоли, а уже потом считывать файл для записи.

Задание 13.2. Введите с консоли имя файла. Найдите максимальный байт в файле, выведите его на экран. Если файла с введенным именем не существует, то программа должна вывести надпись «Файл не существует.», еще раз прочитать имя файла с консоли, а уже потом считывать данные из файла.

Задание 13.3. Введите с консоли имя файла. Найдите байт или байты с максимальным количеством повторов. Выведите их на экран через пробел.

Задание 13.4. Введите с консоли имя файла. Найдите минимальный байт в файле, вывдите его на экран. Если файла с введенным именем не существует, то программа должна вывести надпись «Файл не существует.», еще раз прочитать имя файла с консоли, а уже потом считывать данные из файла.

Задание 13.5. Введите с консоли имя файла. Считайте все байты из файла и, не учитывая повторений, отсортируйте их по байт-коду в возрастающем порядке. Выведите отсортированные байты. Пример байтов входного файла следующий: 44 83 44. Пример вывода следующий: 44 83.

Задание 13.6. Введите с консоли имя файла. Посчитайте в файле количество запятых (т. е. символов ','), количество выведите на консоль. Нужно сравнивать с ASCII-кодом символа ','.

Задание 13.7. Считайте с консоли три имени файла: *файл1*, *файл2*, *файл3*. Разделите *файл1* по следующему критерию: первую половину байт запишите в *файл2*, вторую половину байт запишите в *файл3*. Если в *файл1* количество байт нечетное, то *файл2* должен содержать большую часть.

Задание 13.8. Считайте с консоли два имени файла: *файл1*, *файл2*. Запишите в *файл2* все байты из *файл1*, но в обратном порядке.

Задание 13.9. Введите с консоли имя файла. Посчитайте в файле количество букв английского алфавита, количество выведите на консоль.

Задание 13.10. Считайте с консоли три имени файла. Запишите в первый файл содержимое второго файла, а потом допишите в первый файл содержимое третьего файла.

Задание 13.11. Считайте с консоли два имени файла. В начало первого файла запишите содержимое второго файла так, чтобы получилось объединение файлов.

Задание 13.12. Считайте с консоли два имени файла. Выведите во второй файл все байты с четным индексом.

Задание 13.13. Считайте с консоли два имени файла. Выведите во второй файл все числа, которые есть в первом файле. Числа выводите через пробел.

Пример тела первого файла следующий: 12 text var2 14 8v 1. Результат во втором файле следующий: 12 14 1.

Задание 13.14. Считайте с консоли два имени файла. Выведите во второй файл все содержимое первого файла, заменив все точки на восклицательный знак.

Задание 13.15. В папке *theme13* создайте папку (package) с именем *exercise13_2*.

В папке *exercise13_2* создайте класс *Student* (студент), который реализует интерфейс *Serializable*.

В классе *Student* создайте следующие переменные: *id* (номер зачетной книжки) типа *String*, *name* (ФИО) типа *String*, *year* (курс) типа *int*, *faculty* (факультет) типа *String*. Первые три переменные приватные, последняя переменная – с модификатором доступа в пределах пакета.

В классе *Student* создайте параметризованный конструктор, позволяющий инициализировать первые три переменные.

В классе *Student* переопределите метод *toString*( ), чтобы выводить все сведения о студенте.

В папке *exercise13_2* создайте класс *DemoSerialization*.

В классе *DemoSerialization* создайте метод *main*.

В методе *main* инициализируйте переменную *faculty* значением «ФЭУ».

В методе *main* создайте переменную *bestStudent* как экземпляр класса (объекта) *Student*.

В методе *try* с ресурсами создайте поток *ObjectOutputStream* для записи в файл *demoStud.txt* сведений о студенте. Запишите в поток сведения о студенте *bestStudent* с помощью метода *writeObject*( ). Обработайте исключение *IOException*.

Измените значение статической переменной *faculty* на «УФФ».

В методе *try* с ресурсами создайте поток *ObjectInputStream* для чтения сведений о студенте из файла *demoStud.txt*. Прочтите эти сведения с помощью метода *readObject*( ). Выведите эти сведения на консоль. Обработайте по отдельности исключения *ClassNotFoundException*, *FileNotFoundException* и *IOException*.

Задание 13.16. На основании рисунка 14 восстановите фрагменты кода, чтобы получилась рабочая программа на языке Java, которая выдаст следующий результат:

12

8

Возможно, понадобятся не все фрагменты. Один фрагмент можно использовать несколько раз. Добавьте недостающие фигурные скобки.

Рисунок 14 – **Части программы к заданию 13.16**

14. КЛАССЫ-КОЛЛЕКЦИИ

14.1. Понятие коллекции

Для хранения большого количества однотипных данных могут использоваться массивы, но они не всегда являются идеальным решением. Во-первых, длина массива задается заранее и в случае, если количество элементов заранее неизвестно, придется выделять память «с запасом» либо предпринимать сложные действия по переопределению массива. Во-вторых, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.

В программировании давно и эффективно используются такие структуры данных как стек, очередь, список, множество и т. д., объединенные общим названием коллекция. *Коллекция* – это группа элементов с операциями добавления, извлечения и поиска элемента. Механизм работы операций существенно различается в зависимости от типа коллекции. Например, элементы *стека* упорядочены в последовательность, добавление нового элемента может происходить только в конец этой последовательности, получить можно только элемент, находящийся в конце (добавленный последним). *Очередь* позволяет получить лишь первый элемент (элементы добавляются в один конец последовательности, а «забираются» с другого). Другие коллекции (например, *список*) позволяют получить элемент из любого места последовательности, а *множество* вообще не упорядочивает элементы и позволяет (помимо добавления и удаления) только узнать, содержится ли в нем данный элемент.

Существуют многочисленные альтернативы Java Collections Framework. Библиотека стандартных коллекций собрана в пакете *java.util*.

В библиотеке коллекций Java существуют следующие базовые интерфейсы, реализации которых и представляют совокупность всех классов коллекций:

- *Collection* – коллекция, которая содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (*add, addAll*), удаление (*remove, removeAll, clear*), поиск (*contains*).
- *Map* описывает коллекцию, состоящую из пар «ключ – значение». У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (*map*). Такую коллекцию часто называют еще словарем (*dictionary*) или ассоциативным массивом (*associative array*), которая не относится к интерфейсу *Collection* и является самостоятельной.

Хотя фреймворк называется *Java Collections Framework*, интерфейс *map* и его реализации тоже входят в фреймворк.

14.2. Интерфейс *Collection*

Интерфейс *Collection* является обобщенным и объявляется следующим образом:

```
interface Collection<E>
```

E обозначает тип объектов, которые будет содержать коллекция.

Интерфейс *Collection* расширяет интерфейс *Iterable*, у которого есть только один метод *iterator*( ). Это значит что любая коллекция, которая является наследником *Iterable*, должна возвращать итератор.

Итератор – объект, абстрагирующий за единым интерфейсом доступ к элементам коллекции. Итератор – это паттерн, позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации. Это означает, что все коллекции можно перебирать, организовав цикл *for* в стиле *for-each*.

В интерфейсе *Collection* определяются основные методы, которые должны иметь все коллекции. Эти методы перечислены в таблице 15.

Таблица 15 – **Методы из интерфейса *Collection***

Метод	Описание
boolean add(E объект)	Вводит заданный объект в вызывающую коллекцию. Возвращает логическое значение *true*, если объект успешно введен в коллекцию. Если объект уже присутствует в коллекции, которая не допускает дублирование объектов, то возвращается логическое значение *false*
boolean addAll(Collection<? extends E> c)	Вводит все элементы заданной коллекции *c* в вызывающую коллекцию. Возвращает логическое значение *true*, если коллекция изменена (т. е. все элементы введены), в противоположном случае – логическое значение *false*
void clear( )	Удаляет все элементы из вызывающей коллекции
boolean contains(Object объект)	Возвращает логическое значение *true*, если заданный объект является элементом вызывающей коллекции, в противоположном случае – логическое значение *false*
boolean containsAll (Collection< ? > c)	Возвращает логическое значение *true*, если вызывающая коллекция содержит все элементы заданной коллекции *c*, в противоположном случае – логическое значение *false*
boolean equals(Object объект)	Возвращает логическое значение *true*, если вызывающая коллекция и заданный объект равнозначны, в противоположном случае – логическое значение *false*
int hashCode( )	Возвращает хеш-код вызывающей коллекции
boolean isEmpty( )	Возвращает логическое значение *true*, если вызывающая коллекция пуста, в противоположном случае – логическое значение *false*

Продолжение таблицы 15

Метод	Описание
Iterator<E> iterator( )	Возвращает итератор для вызывающей коллекции
default Stream<E> parallelStream( )	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются параллельные операции ввода-вывода, если это вообще возможно (добавлен в версии J DK 8)
boolean remove(Object объект)	Удаляет один экземпляр объекта из вызывающей коллекции. Возвращает логическое значение *true*, если элемент удален, в противоположном случае – логическое значение *false*
boolean removeAll(Collection< ? > c)	Удаляет все элементы заданной коллекции *c* из вызывающей коллекции. Возвращает логическое значение *true*, если в конечном итоге коллекция изменяется (т. е. элементы из нее удалены), в противоположном случае – логическое значение *false*
default boolean removeIf(Predicate< ? super E > предикат)	Удаляет из вызывающей коллекции элементы, удовлетворяющие условию, которое задает предикат (добавлен в версии JDK 8)
boolean retainAll(Collection< ? > c)	Удаляет из вызывающей коллекции все элементы, кроме элементов заданной коллекции *c*. Возвращает логическое значение *true*, если в конечном итоге коллекция изменяется (т. е. элементы из нее удалены), в противоположном случае – логическое значение *false*
int size( )	Возвращает количество элементов, содержащихся в коллекции
default Spliterator<E> spliterator( )	Возвращает итератор-разделитель для вызывающей коллекции (добавлен в версии JDK 8)
Default Stream<E> stream( )	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются последовательные операции ввода-вывода (добавлен в версии JDK 8)
Object[ ] toArray( )	Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции

Окончание таблицы 15

Метод	Описание
<T> T[ ] toArray(T массив [ ])	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер заданного массива равен количеству элементов в коллекции, они возвращаются в этом массиве. Если же размер массива меньше количества элементов в коллекции, то создается и возвращается новый массив нужного размера. Если размер массива больше количества элементов в коллекции, то во всех элементах, следующих за последним из коллекции, устанавливается пустое значение *null*. Если любой элемент коллекции относится к типу, не являющемуся подтипом массива, то генерируется исключение типа *ArrayStoreException*

Интерфейс *Collection* расширяют интерфейсы *List*, *Set* и *Queue*.

14.3. Интерфейс *List*

Представляет собой упорядоченную коллекцию, в которой допустимы дублирующие значения. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. Интерфейс *List* является обобщенным и объявляется следующим образом:

```
interface List<E>
```

Параметр E обозначает тип объектов, которые должен содержать список.

Помимо методов, объявленных в интерфейсе *Collection*, в интерфейсе *List* определяется ряд своих методов (таблица 16).

Таблица 16 – **Некоторые методы из интерфейса *List***

Метод	Описание
void add(int индекс, E объект)	Вводит заданный объект на позиции вызывающего списка по указанному индексу. Любые введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются

Окончание таблицы 16

Метод	Описание
boolean addAll(int индекс, Collection<? Extends E> c)	Вводит все элементы из коллекции *c* в вызывающий список, начиная с позиции по указанному индексу. Введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются. Возвращает логическое значение *true*, если вызывающий список изменяется, в противоположном случае – логическое значение *false*
E qet(int индекс)	Возвращает объект, хранящийся в вызывающем списке, на позиции по указанному индексу
int indexOf(Object объект)	Возвращает индекс первого экземпляра заданного объекта в вызывающем списке. Если заданный объект отсутствует в списке, возвращается значение –1
int lastIndexOf(Object объект)	Возвращает индекс последнего экземпляра заданного объекта в вызывающем списке. Если заданный объект отсутствует в списке, возвращается значение –1
E remove(int индекс)	Удаляет элемент из вызывающего списка на позиции по указанному индексу и возвращает удаленный элемент. Результирующий список уплотняется, т. е. элементы, следующие за удаленным, смещаются на одну позицию назад
List<E> subList( ) (int начало, int конец)	Возвращает список, включающий элементы от позиции начало до позиции конец –1 из вызывающего списка. Ссылки на элементы из возвращаемого списка сохраняются и в вызывающем списке

Варианты методов *add*( ) и *addAll*( ), определенные в интерфейсе *Collection*, дополняются в интерфейсе *List* методами *add*(*int n, E*) и *addAll*(*int n, Collection*). Эти методы вводят элементы на позиции по указанному индексу *n*. Кроме того, методы *add*(*E*) и *addAll*(*Collection*), определенные в интерфейсе *Collection*, в интерфейсе *List* вводят элементы в конец списка.

14.4. Интерфейс *Set*

Этот интерфейс расширяет интерфейс *Collection* и описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. В этом интерфейсе не определяется никаких дополнительных методов. Интерфейс *Set* является обобщенным и объявляется следующим образом:

```
interface Set<E>
```

E обозначает тип объектов, которые должно содержать множество. Расширяет интерфейс *Set* интерфейс *SortedSet*, который определяет поведение множеств, отсортированных в порядке возрастания. Расширяет интерфейс *SortedSet* интерфейс *NavigableSet*, который определяет поведение коллекции, извлечение элементов из которой осуществляется на основании наиболее точного совпадения с заданным значением или несколькими значениями.

14.5. Интерфейс *Queue*

Этот интерфейс расширяет интерфейс *Collection* и определяет поведение очереди, которая действует как список по принципу «первым вошел – первым обслужен». Существуют разные виды очередей (не менее 1 см), порядок организации в которых основывается на некотором критерии. Интерфейс *Queue* является обобщенным и объявляется следующим образом:

```
interface Queue<E>
```

E обозначает тип объектов, которые будут храниться в очереди.

Расширяет интерфейс *Queue* интерфейс *Dequeue*, который определяет поведение двусторонней очереди, которая может функционировать как стандартная очередь по принципу «первым вошел – первым обслужен» или как стек по принципу «последним вошел – первым обслужен».

14.6. Интерфейс *Map*

Описывает коллекцию, состоящую из пар «ключ – значение». У каждого ключа только одно значение. Такую коллекцию часто назы-

вают еще словарем (*dictionary*) или ассоциативным массивом (*associative array*). Она не относится к интерфейсу *Collection* и является самостоятельной.

Ключ – это объект, используемый для последующего извлечения данных. Задавая ключ и значение, можно размещать значение в отображении, представленном объектом типа *Map*. Сохранив значение по ключу, можно получить его обратно по этому же ключу. Интерфейс *Map* является обобщенным и объявляется следующим образом:

```
interface Map<K, V>
```

K обозначает тип ключей, V – тип хранимых значений.

Некоторые наиболее часто употребляемые методы интерфейса *Map* приведены в таблице 17.

Таблица 17 – **Методы из интерфейса *Map***

Метод	Описание
void clear( )	Удаляет все пары «ключ – значение» из вызывающего отображения
boolean containsKey (Object k)	Возвращает логическое значение *true*, если вызывающее отображение содержит указанный ключ *k*, в противоположном случае – логическое значение *false*
boolean containsValue (Object v)	Возвращает логическое значение *true*, если вызывающее отображение содержит значение *v*, в противоположном случае – логическое значение *false*
Set<Map.Ent:ry<K, V>> entrySet( )	Возвращает множество типа *Set*, содержащее все записи из вызывающего отображения в виде объектов типа *Map.Entry*. Следовательно, этот метод возвращает представление вызывающего отображения в виде множества
V qet(Object k)	Возвращает значение, связанное с указанным ключом *k*. Если же ключ не найден, то возвращается пустое значение *null*
boolean isEmpty( )	Возвращает логическое значение *true*, если вызывающее отображение пусто, в противоположном случае – логическое значение *false*
Set<K>keySet( )	Возвращает множество, содержащее ключи из вызывающего отображения. Следовательно, этот метод возвращает представление ключей в вызывающем отображении в виде множества

Окончание таблицы 17

Метод	Описание
V put(K k, V v)	Вводит новое значение *v*, вызывающее отображение, перезаписывая любое предшествующее значение, связанное с заданным ключом *k*. Возвращает пустое значение *null*, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом
V remove(Object k)	Удаляет запись по заданному ключу *k*
int size( )	Возвращает количество пар «ключ – значение» в вызывающем отображении
Collection<V>values( )	Возвращает коллекцию, содержащую значения из вызывающего отображения

14.7. Класс *ArrayList*

Класс *ArrayList* расширяет класс *AbstractList* и реализует интерфейс *List*. Класс *ArrayList* является обобщенным и объявляется следующим образом:

```
class ArrayList<E>
```

Параметр E обозначает тип сохраняемых объектов.

В классе *ArrayList* поддерживаются динамические массивы, которые могут наращиваться по мере надобности. Стандартные массивы в Java имеют фиксированную длину. После того как массив создан, он не может увеличиваться или уменьшаться, а следовательно, нужно заранее знать, сколько элементов требуется в нем хранить. Иногда еще до стадии выполнения неизвестно, насколько большой массив потребуется. В качестве выхода из данного положения в каркасе коллекций определяется класс *ArrayList*. Класс *ArrayList* представляет собой списочный массив объектных ссылок переменной длины. Это означает, что размер объекта типа *ArrayList* может динамически увеличиваться или уменьшаться. Списочные массивы создаются с некоторым начальным размером. Когда же этого первоначального размера оказывается недостаточно, коллекция автоматически расширяется. Когда из коллекции удаляются объекты, она может сокращаться.

В классе *ArrayList* определены следующие конструкторы:

```
ArrayList ( )
ArrayList (Collection < ? extends E > c)
ArrayList (int емкость)
```

Первый конструктор создает пустой списочный массив, второй – списочный массив, инициализируемый элементами из заданной коллекции *c*, а третий – списочный массив, имеющий начальную емкость. Под емкостью здесь подразумевается размер базового массива, используемого для хранения элементов данного вида коллекции. Емкость наращивается автоматически по мере ввода элементов в списочный массив.

14.8. Класс *HashSet*

Класс *HashSet* расширяет класс *AbstractSet* и реализует интерфейс *Set*. Он служит для создания коллекции, для хранения элементов которой используется хеш-таблица. Класс *HashSet* является обобщенным, объявляется следующим образом:

```
class HashSet<E>
```

Параметр E обозначает тип объектов, которые будут храниться в хеш-множестве.

В классе *HashSet* определены следующие конструкторы:

```
HashSet ( )
HashSet (Collection<? extends E > c)
HashSet (int емкость)
HashSet (int емкость , float коэффициент_заполнения)
```

В первой форме конструктора хеш-множество создается по умолчанию. Во второй форме хеш-множество инициируется содержимым заданной коллекции *c*. В третьей форме задается емкость хеш-множества (по умолчанию – 16), а в четвертой в качестве аргументов конструктора задается емкость хеш-множества и коэффициент заполнения, называемый емкостью загрузки. Коэффициент заполнения должен быть в пределах от 0,0 до 1,0. Данный показатель определяет, насколько заполненным должно быть хеш-множество, прежде чем будет изменен его размер. В частности, когда количество элементов становится больше емкости хеш-множества, умноженной на коэффи-

циент заполнения, такое хеш-множество расширяется. В конструкторах, которые не принимают коэффициент заполнения в качестве параметра, выбирается значение этого коэффициента, равное 0,75.

В классе *HashSet* не определяется никаких дополнительных методов, помимо тех, что предоставляют его суперклассы и интерфейсы. Следует также иметь в виду, что класс *HashSet* не гарантирует упорядоченности элементов. Если же требуются сортированные множества, то для этой цели лучше выбрать другой вид коллекции, например *TreeSet*.

14.9. Доступ к коллекциям через итератор

Нередко требуется перебрать все элементы коллекции, например, вывести каждый ее элемент. Для этого можно, например, воспользоваться итератором – объектом класса, который реализует один из двух интерфейсов (*Iterator* или *ListIterator*). В частности, интерфейс *Iterator* позволяет организовать цикл для перебора коллекции, извлекая или удаляя из нее элементы. Интерфейс *ListIterator* расширяет интерфейс *Iterator* для двустороннего обхода списка и видоизменения его элементов. Интерфейсы *Iterator* и *ListIterator* являются обобщенными и объявляются следующим образом:

```
interface Iterator<E>
interface ListIterator<E>
```

Параметр E обозначает тип перебираемых объектов.

Для применения итератора для перебора содержимого коллекции нужно следующее:

- установить итератор на начало коллекции, получив его из метода *iterator*( ), вызываемого для коллекции;
- организовать цикл, в котором вызывается метод *hasNext*( ); перебирать содержимое коллекции до тех пор, пока метод *hasNext*( ) не возвратит логическое значение *false*;
- получить в цикле каждый элемент коллекции, вызывая метод *next*( ).

Вывод на экран элементов *Set* с помощью итератора можно представить следующим образом:

```
public static void main(String[ ] args)
{
```

```
      Set<String> set = new HashSet<String>();
      set.add("A");
      set.add("B");
      set.add("C");

      //получение итератора для множества
      Iterator<String> iterator = set.iterator();

      while (iterator.hasNext()) //проверка, есть ли еще эле-
менты
      {
         //получение текущего элемента и переход на следующий
         String text = iterator.next();

         System.out.println(text);
      }
   }
```

Если не требуется видоизменять содержимое коллекции или извлекать из нее элементы в обратном порядке, в таком случае можно использовать цикл *for* в стиле *for-each*. Так как все классы коллекций реализуют интерфейс *Iterable*, то ими можно оперировать в цикле *for*. Предыдущий пример с циклом *for-each* будет иметь более компактную следующую запись:

```
   public static void main(String[ ] args)
   {
      Set<String> set = new HashSet<String>();
      set.add("A");
      set.add("B");
      set.add("C");

      for (String text : set)
      {
         System.out.println(text);
      }
   }
```

Отображения *Map* не реализуют интерфейс *Iterable*. Это означает, что перебрать содержимое отображения, организовав цикл *for* в стиле

for-each нельзя, нельзя получить итератор отображения. Можно получить представление отображения в виде коллекции, которое допускает перебор содержимого в цикле или с помощью итератора.

14.10. Класс *HashMap*

Этот класс расширяет класс *AbstractMap* и реализует интерфейс *Map*. В нем используется хеш-таблица для хранения отображения, благодаря этому обеспечивается постоянное время выполнения методов *get*( ) и *put*( ) даже в обращении к крупным отображениям. Класс *HashMap* является обобщенным и объявляется следующим образом:

```
class HashMap<K , V>
```

Параметр K обозначает тип ключей, V – тип хранимых в отображении значений.

В классе определены следующие конструкторы:

```
HashMap ( )
HashMap (Map<? extends K, ? extends V> m)
HashMap (int емкость)
HashMap (int емкость, float коэффициент_заполнения)
```

Вывод на экран элементов *Map* можно представить следующим образом:

```
public static void main(String[ ] args)
{
   //все элементы хранятся в парах
   Map<String,   String>   map   =   new   HashMap<String,
String>();
   map.put("first", "A");
   map.put("second", "B");
   map.put("third", "C");

   Iterator<Map.Entry<String,  String>>  iterator  =  map.
entrySet().iterator();

   while (iterator.hasNext())
   {
```

```
        //получение «пары» элементов
        Map.Entry<String, String> pair = iterator.next();
        String key = pair.getKey(); //ключ
        String value = pair.getValue(); //значение
        System.out.println(key + ":" + value);
    }
}
```

Задания для самостоятельной работы

Задание 14.1. Введите с клавиатуры 20 целых чисел и сохраните их в списке. Создайте следующий метод по безопасному извлечению чисел из следующего списка: `int safeGetElement(ArrayList<Integer> list, int index, int defaultValue)`. Метод должен возвращать элемент списка *list* по его индексу *index*. Если в процессе получения элемента возникло исключение, его нужно перехватить, метод должен вернуть *defaultValue*.

Задание 14.2. Введите с клавиатуры в список 20 слов. Подсчитайте количество одинаковых слов в списке. Результат представьте в виде словаря *Map<String, Integer>*, в котором первый параметр – уникальная строка, а второй – число (сколько раз данная строка встречалась в списке). Выведите содержимое словаря на экран.

Тема 15. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

15.1. Создание потока

В последовательном программировании все действия, выполняемые программой, выполняются друг за другом. Для многих задач удобно и даже необходимо организовать параллельное выполнение нескольких частей программы. Каждая из таких самостоятельных задач называется потоком (thread).

Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

Существуют следующие способы создания потока:

- реализация интерфейса *Runnable*;
- наследование класса *Thread*.

15.2. Реализация интерфейса *Runnable*

Самый простой способ создания потока заключается в определении класса, который реализует *Runnable*. Интерфейс *Runnable* определяет всего один абстрактный метод – *run*( ).

В теле метода *run*( ) реализуется работа потока (вызываются классы, методы, определяются переменные). При выходе из метода *run*( ) поток завершает свое действие.

Класс, реализующий интерфейс *Runnable* выглядит следующим образом:

```
class MyClass implements Runnable {
   public void run() {
      // тело метода run
   }
}
```

Для создания потока необходимо создать следующий экземпляр класса, реализующего интерфейс *Runnable*:

```
Runnable r = new MyClass();
```

На основе объекта *Runnable* необходимо создать следующий объект *Thread*:

```
Thread t = new Thread(r);
```

Для того чтобы запустить поток, необходимо создать метод *start*( ) класса *Thread* для данного потока. Данный метод запускает метод *run*( ).

Пример реализации интерфейса *Runnable* следующий:

```
class NewThread implements Runnable {
   public void run() {
      System.out.println("Тело метода run() созданного по-
тока.");

   }
}
```

```
public class Main {
   public static void main(String args[ ]) {
      System.out.println("Основной поток.");
      Runnable r = new NewThread();
      Thread t = new Thread(r);
      t.start();
   }
}
/* результат:
Основной поток.
Тело метода run() созданного потока.
*/
```

15.3. Наследование класса *Thread*

Для создания потока необходимо расширить класс *Thread* и переопределить метод *run*( ). Как и в случае с реализацией интерфейса в теле метода *run*( ) реализуется работа потока, при выходе из метода поток прекращает свою работу.

Класс, расширяющий *Thread*, выглядит следующим образом:

```
class MyClass extends Thread {
   public void run() {
      // тело метода run()
   }
}
```

Пример расширения класса *Thread* следующий:

```
class AppThread extends Thread {
   public void run() {

      System.out.println("Дочерний поток.");
      for(int i = 1; i <= 5; i++) {
         System.out.println("Значение цикла дочернего пото-
ка - " + i);
      }
```

```
        System.out.println("Работа дочернего потока заверше-
на.");
      }
   }
   public class App {
      public static void main(String[ ] args) {
         System.out.println("Родительский поток.");
         Thread t = new Thread(new NewThread());
         t.start();
      }
   }/*результат:
   Родительский поток.
   Дочерний поток.
   Значение цикла дочернего потока - 1
   Значение цикла дочернего потока - 2
   Значение цикла дочернего потока - 3
   Значение цикла дочернего потока - 4
   Значение цикла дочернего потока - 5
   Работа дочернего потока завершена.
   */
```

Реализация интерфейса *Runnable* используется в случаях, когда класс уже наследует какой-либо родительский класс, тем самым не позволяет расширить класс *Thread*. Расширение класса *Thread* целесообразно используется, когда необходимо переопределить другие методы класса *Thread*, помимо метода *run*( ). Это используется довольно редко.

Класс *Thread* имеет семь перегруженных конструкторов:

```
Thread();
Thread(Runnable target);
Thread(Runnable target, String name);
Thread(String name);
Thread(ThreadGroup group, Runnable target);
Thread(ThreadGroup group, Runnable target, String name);
Thread(ThreadGroup group, String name);
```

Target – экземпляр класса, реализующего интерфейс *Runnable*; *name* – имя создаваемого потока; *group* – группа, к которой относится данный поток.

Например, создадим поток, который будет входить в группу, реализовывать интерфейс *Runnable* и иметь свое уникальное название:

```
Runnable r = new MyClassRunnable(); // в данном классе создается поток
ThreadGroup tg = new ThreadGroup(); // создаем группу потоков
Thread t = new Thread(tg, r, “myThread”); // создаем экземпляр класса потока
```

Группы потоков удобно использовать, когда необходимо одинаково управлять несколькими потоками. Например, есть потоки, которые выводят данные на печать, необходимо прервать печать всех документов, поставленных в очередь. В этом случае удобно применить команду к группе потока, а не к каждому потоку отдельно. Это можно сделать, если потоки отнесены к одной группе.

Для того чтобы запустить поток, необходимо вызвать следующий метод *start*( ):

```
Thread t = new Thread();
t.start();
```

Если вместо метода *start*( ) выполнить метод *run*( ), то *run*( ) выполнит свой код, но только в том же потоке, в котором и был вызван. Отдельный поток при этом не создастся.

Для управления потоком класс *Thread* предоставляет еще ряд методов. К наиболее используемые из них относятся следующие:

- *getName*( ) возвращает имя потока.
- *setName*(*String name*) устанавливает имя потока.
- *getPriority*( ) возвращает приоритет потока.
- *setPriority*(*int proirity*) устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из ряда потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета от 1 до 10. По умолчанию главному потоку выставляется средний приоритет – 5.
- *isAlive*( ) возвращает *true*, если поток активен.
- *isInterrupted*( ) возвращает *true*, если поток был прерван.
- *join*( ) ожидает завершение потока.
- *run*( ) определяет точку входа в поток.

• *sleep*( ) приостанавливает поток на заданное количество миллисекунд.

• *start*( ) запускает поток, вызывая его метод *run*( ).

Поток может находиться в одном из следующих состояний: выполняющимся; готовом к выполнению, как только он получит время и ресурсы центрального процессора; приостановленном (временно не выполняющимся); возобновленном в дальнейшем; заблокированном в ожидании ресурсов для своего выполнения; завершенном (выполнение закончено и не может быть возобновлено).

15.4. Синхронизация потоков. Оператор *synchronized*

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми. Например, определим следующий код:

```
public class ThreadsApp {
   public static void main(String[ ] args) {
      CommonResource commonResource= new CommonResource();
      for (int i = 1; i < 6; i++){
         Thread t = new Thread(new CountThread(commonResource));
         t.setName("Поток "+ i);
         t.start();
      }
   }
}
class CommonResource{
   int x=0;
}
class CountThread implements Runnable{
   CommonResource res;
   CountThread(CommonResource res){
      this.res=res;
   }
   public void run(){
      res.x=1;
      for (int i = 1; i < 5; i++){
```

```
            System.out.printf("%s %d \n", Thread.currentThread
().getName(), res.x);
            res.x++;
            try{
               Thread.sleep(100);
            }
            catch(InterruptedException e){}
         }
      }
   }
```

Здесь определен класс *CommonResource*, который представляет общий ресурс. В нем определено одно целочисленное поле *x*. Этот ресурс используется классом потока *CountThread*. Класс *CountThread* просто увеличивает в цикле значение *x* на единицу, причем при входе в поток значение *x* = 1. Можно ожидать, что после выполнения цикла *res.x* будет равно 4.

В главном классе программы запускается пять потоков. Можно ожидать, что каждый поток будет увеличивать *res.x* с 1 до 4 и так пять раз. Результат работы программы будет следующим:

```
Поток 1 1
Поток 2 1
Поток 3 1
Поток 5 1
Поток 4 1
Поток 5 6
Поток 2 6
Поток 1 6
Поток 3 6
Поток 4 6
Поток 4 11
Поток 2 11
Поток 5 11
Поток 3 11
Поток 1 11
Поток 4 16
Поток 1 16
Поток 3 16
Поток 5 16
Поток 2 16
```

Пока один поток не окончил работу с полем *res.x*, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова *synchronized*.

Главным для синхронизации в Java является понятие монитора, контролирующего доступ к объекту. Монитор реализует принцип блокировки. Если объект заблокирован одним потоком, то он оказывается недоступным для других потоков. В какой-то момент объект разблокируется, благодаря чему другие потоки смогут получить к нему доступ. Синхронизировать код можно двумя способами. В обоих используется ключевое слово *synchronized*. Этот оператор *synchronized* предваряет блок кода или метод, который подлежит синхронизации. Для его применения необходимо изменить следующий класс *CountThread*:

```
   class CountThread implements Runnable{
      CommonResource res;
      CountThread(CommonResource res){
         this.res=res;
      }
      public void run(){
         synchronized(res){
            res.x=1;
            for (int i = 1; i < 5; i++){
               System.out.printf("%s  %d  \n",  Thread.current
Thread().getName(), res.x);
               res.x++;
               try{
                  Thread.sleep(100);
               }
               catch(InterruptedException e){}
            }
         }
      }
   }
```

При создании синхронизированного блока кода после оператора *synchronized* идет объект-заглушка *synchronized*(*res*). В качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа. Когда выполнение кода доходит до оператора

synchronized, монитор объекта *res* блокируется. На время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, монитор объекта *res* освобождается и становится доступным для других потоков. После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

Если оператор *synchronized* применяется к методу, то когда такой метод получает управление, вызывающий поток активизирует монитор, что приводит к блокированию объекта. Если объект блокирован, он недоступен из другого потока, а кроме того, его нельзя вызвать из других синхронизированных методов, определенных в классе данного объекта. Когда выполнение синхронизированного метода завершается, монитор разблокирует объект, что позволяет другому потоку использовать этот метод.

Для применения *synchronized* к методу необходимо изменить следующие классы программы:

```
   public class ThreadsApp {
      public static void main(String[ ] args) {
         CommonResource commonResource= new CommonResource();
         for (int i = 1; i < 6; i++){
            Thread  t  =  new  Thread(new  CountThread(common-
Resource));
            t.setName("Поток "+ i);
            t.start();
         }
      }
   }
   class CommonResource{
      int x;
      synchronized void increment(){
         x=1;
         for (int i = 1; i < 5; i++){
            System.out.printf("%s   %d   \n",   Thread.current-
Thread().getName(), x);
            x++;
            try{
               Thread.sleep(100);
            }
```

```
            catch(InterruptedException e){}
         }
      }
   }
   class CountThread implements Runnable{
      CommonResource res;
      CountThread(CommonResource res){
         this.res=res;
      }
      public void run(){
         res.increment();
      }
   }
```

Результат работы в данном случае будет аналогичен примеру выше с блоком *synchronized*. Здесь опять в дело вступает монитор объекта *CommonResource* (общего объекта для всех потоков). Поэтому синхронизированным объявляется не метод *run*( ) в классе *CountThread*, а метод *increment* класса *CommonResource*. Когда первый поток начинает выполнение метода *increment*, он захватывает монитор объекта *CommonResource*. Все потоки также продолжают ожидать его освобождения.

15.5. Организация взаимодействия потоков с помощью методов *notify*( ), *wait*( ) и *notifyAll*( )

В качестве примера необходимо рассмотреть следующую ситуацию. Поток *T*, который выполняется в синхронизированном методе, нуждается в доступе к ресурсу *R*, который временно недоступен. Необходимо определить, что делать потоку *T*. Начать выполнение цикла опросов в ожидании того момента, когда освободится ресурс *R*. Тогда поток *T* будет связывать объект, препятствуя доступу к нему других потоков. Такое решение малопригодно, поскольку оно сводит на нет все преимущества программирования в многопоточной среде. Будет гораздо лучше, если поток *T* временно разблокирует объект и позволит другим потокам воспользоваться его методами. Когда ресурс *R* станет доступным, поток *T* получит об этом уведомление и возобновит свое исполнение. Для того чтобы такое решение можно было реализовать, необходимы средства взаимодействия потоков, с помощью

которых один поток мог бы сообщить другому потоку о том, что он приостановил свое исполнение, а также получить уведомление о том, что его исполнение может быть возобновлено.

Для организации подобного взаимодействия потоков в классе *Object* предусмотрены следующие методы:

- *wait*( ) освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод *notify*( );
- *notify*( ) продолжает работу потока, у которого ранее был вызван метод *wait*( );
- *notifyAll*( ) возобновляет работу всех потоков, у которых ранее был вызван метод *wait*( ).

Все эти методы вызываются только из синхронизированного контекста – синхронизированного блока или метода.

Рассмотрим, как можно использовать эти методы. Необходимо взять следующую стандартную задачу «Производитель – Потребитель» (“Producer – Consumer”): пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. При этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем следующие методы *wait*( ) и *notify*( ):

```
public class ThreadsApp {
   public static void main(String[ ] args) {
Store store=new Store();
     Producer producer = new Producer(store);
     Consumer consumer = new Consumer(store);
     new Thread(producer).start();
     new Thread(consumer).start();
   }
}
// Класс Магазин, хранящий произведенные товары
class Store{
   private int product = 0;
   public synchronized void get() {
      while (product < 1) {
         try {
            wait();
         }
         catch (InterruptedException e) {
```

```
        }
      }
      product--;
      System.out.println("Покупатель купил 1 товар");
      System.out.println("Товаров на складе: " + product);
      notify();
   }
   public synchronized void put() {
      while (product >= 3) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product++;
      System.out.println("Производитель добавил 1 товар");
      System.out.println("Товаров на складе: " + product);
      notify();
   }
}
// класс Производитель
class Producer implements Runnable{

   Store store;
   Producer(Store store){
      this.store=store;
   }
   public void run(){
      for (int i = 1; i < 6; i++) {
         store.put();
      }
   }
}
// Класс Потребитель
class Consumer implements Runnable{

   Store store;
   Consumer(Store store){
      this.store = store;
   }
```

```
    public void run(){
      for (int i = 1; i < 6; i++) {
        store.get();
      }
    }
}
```

Здесь определен класс магазина, потребителя и покупателя. Производитель в методе *run*( ) добавляет в объект *Store* с помощью его метода *put*( ) 6 товаров. Потребитель в методе *run*( ) в цикле обращается к методу *get* объекта *Store* для получения этих товаров. Оба метода класса *Store* (*put* и *get*) являются синхронизированными.

Для отслеживания наличия товаров в классе *Store* необходимо проверить значение переменной *product*. По умолчанию товара нет, поэтому переменная равна 0. Метод *get*( ) (получение товара) должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе *get* в бесконечном цикле необходимо проверить, отсутствует ли товар:

```
while (product < 1)
```

Если товар отсутствует, вызывается метод *wait*( ). Этот метод освобождает монитор объекта *Store* и блокирует выполнение метода *get*, пока для этого же монитора не будет вызван метод *notify*( ).

Когда в методе *put*( ) добавляется товар и вызывается *notify*( ), то метод *get*( ) получает монитор и выходит из цикла *while* (*product* < 1), так как товар добавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, уменьшается значение *product*:

```
product--.
```

В конце вызов метода *notify*( ) дает сигнал методу *put*( ) продолжить работу.

В методе *put*( ) используется похожий механизм, только теперь метод *put*( ) должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, если товар уже есть, то необходимо освободить монитор с помощью *wait*( ) и ждать вызова *notify*( ) в методе *get*( ).

Программа покажет следующие результаты:

```
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Производитель добавил 1 товар
Товаров на складе: 3
Покупатель купил 1 товар
Товаров на складе: 2
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
Производитель добавил 1 товар
Товаров на складе: 1
Производитель добавил 1 товар
Товаров на складе: 2
Покупатель купил 1 товар
Товаров на складе: 1
Покупатель купил 1 товар
Товаров на складе: 0
```

Таким образом, с помощью *wait*( ) в методе *get*( ) можно ожидать, когда производитель добавит новый продукт. После добавления можно вызвать *notify*( ), как бы говоря, что магазин теперь снова пуст, можно еще добавлять.

В методе *put*( ) с помощью *wait*( ) можно ожидать освобождения места на складе. После того, как место освободится, необходимо добавить товар и через *notify*( ) уведомить покупателя о том, что он может забирать товар.

Задания для самостоятельной работы

Задание 15.1. В папке *theme14* создайте интерфейс *MusicalInstrument*, наследуемый от *Runnable*. В нем объявите следующие методы: *Date startPlaying*( ) и *Date stopPlaying*( ).

В папке *theme14* создайте класс *Violin* (скрипка), который реализует необходимый интерфейс. Класс должен содержать приватную переменную *owner* типа *String*. Она должна инициализироваться в конструкторе класса с одним параметром.

В методе *startPlaying*( ) на консоль должна быть выведена фраза «owner начинает играть», а затем определяется время начала игры.

В методе *stopPlaying*( ) на консоль должна быть выведена фраза «owner закончил играть», а затем определяется время окончания игры.

В классе *Violin* создайте метод *public static void sleepNSeconds*(*int n*) (где *n* – количество секунд). Метод устанавливает задержку на *n* секунд. Метод не выбрасывает никаких исключений.

В классе *Violin* нужно реализовать еще один метод, в котором сначала считывается время начала игры с помощью метода *startPlaying*( ), затем устанавливается задержка на 1 секунду с помощью метода *sleepNSeconds*( ), потом считывается время окончания игры методом *stopPlaying*( ), на консоль выводится продолжительность игры в миллисекундах.

В папке *theme14* создайте класс *ViolinDemo*.

В классе *ViolinDemo* создайте методе *main*( ). В методе *main*( ) создайте поток для игры на скрипке и запустите его на выполнение.

Задание 15.2. В папке *theme14* создайте класс *Horse* (лошадь), наследуемый от *Thread*. В классе должен быть создан конструктор с одним параметром (кличкой лошади), вызывающий конструктор суперкласса с этим же параметром.

Класс *Horse* содержит одну переменную *isFinished*. Подумайте, какой должен быть тип у этой переменной. Создайте геттер для переменной *isFinished*. Подумайте, какой должен быть модификатор доступа у переменной *isFinished*.

В классе *Horse* нужно реализовать еще один метод, в котором сначала организуется пустой цикл от 0 до 10 000, затем выводится на консоль «horse_name has finished the race!» (используйте метод *getName*( )). Подумайте, что еще нужно добавить в конец этого метода.

В папке *theme14* создайте класс *Exercise*, в котором реализуйте метод *public static List<Horse> prepareHorsesAndStart*( ). В методе создайте список из нескольких (например, из десяти лошадей), запустите на выполнение соответствующие потоки. Кличка каждой лошади состоит из слова «Horse» и порядкового номера лошади.

В классе *Exercise* реализуйте метод *public static int calculateHorsesFinished*(*List<Horse> horses*) *throws InterruptedException*. Он должен подсчитать количество финишировавших лошадей и возвратить его, используя метод *isFinished*( ). Если лошадь еще не пришла к фи-

нишу (! isFinished( )), то следует вывести в консоль “Waiting for horse_name” с указанием клички лошади. Затем нужно подождать, пока она завершит гонку. Подумайте, какой метод нужно использовать для этого.

В методе *main*( ) класса *Exercise* вначале вызовите метод *prepareHorsesAndStart*( ), а затем организуйте забег лошадей, пока они все не завершат гонку. Используйте для этого цикл с проверкой на количество финишировавших лошадей *calculateHorsesFinished*( ).

Задание 15.3. В папке *theme14* создайте класс *Exercise* с двумя статическими переменными *totalCountSpeeches* и *soundsInOneSpeech* типа *int*. Инициализируйте их значениями 200 и 1 000 000 соответственно.

В папке *theme14* создайте класс *Politic* (политик), наследуемый от *Thread*. В классе должен быть создан конструктор с одним параметром (фамилией политика), вызывающий конструктор суперкласса с этим же параметром и запускающий на выполнение соответствующий поток.

В классе *Politic* создайте приватную переменную *countSounds* типа *int*.

В классе *Politic* нужно реализовать метод *getCountSpeaches*( ), который возвращает *countSounds* : *soundsInOneSpeech*.

В классе *Politic* нужно переопределить метод *toString*( ), чтобы он возвращал строку вида «политик сказал речь *n* раз» с указанием фамилии политика. Используйте методы *getName*( ) и *getCountSpeaches*( ).

В классе *Politic* нужно реализовать еще один метод, в котором в цикле значение переменной *countSounds* увеличивается на единицу, пока оно не достигнет значения *totalCountSpeeches* · *soundsInOneSpeech*.

В методе *main*( ) класса *Exercise* вначале определите, что в политической борьбе будут участвовать три политических деятеля (Иванов, Петров и Сидоров). Их политические дебаты будут длиться, пока общее количество речей, произнесенных всеми тремя политиками, не станет равным *totalCountSpeeches*. Для этого организуйте цикл с соответствующей проверкой. Нужно сделать так, чтобы Иванов сказал больше всего речей на политических дебатах. Подумайте, какой метод можно вызвать у объекта *ivanov*, чтобы Иванов разговаривал, пока не завершится все свободное время.

Задание 15.4. В папке *theme14* создайте класс *Kitten* (котенок), наследуемый от *Thread*. В классе должен быть создан конструктор с одним параметром (кличкой котенка), вызывающий конструктор суперкласса с этим же параметром.

В классе *Kitten* реализуйте метод *investigateWorld*( ), заключающийся в том, что котенок засыпает на 200 мс. Метод не выбрасывает никаких исключений.

В классе *Kitten* нужно реализовать еще один метод, в котором на консоль выводится «имя_котенка вылез из корзинки» (с указанием клички котенка), а затем вызывается метод *investigateWorld*( ).

Создайте класс *Cat* (кошка), наследуемый от *Thread*.

Класс *Cat* содержит следующие переменные: *kitten1* и *kitten2* с модификатором доступа *protected*. Подумайте, какой должен быть тип у этих переменных.

В классе *Cat* должен быть создан конструктор с одним параметром (кличкой кошки), вызывающий конструктор суперкласса с этим же параметром. Затем в конструкторе переменные *kitten1* и *kitten2* инициализируются новыми объектами с параметрами «Котенок 1, мама – …» и «Котенок 2, мама – …» соответственно. После этого запускается на выполнение соответствующий поток.

В классе *Cat* должен быть реализован приватный метод *initAllKitten*( ) *throws InterruptedException*, в котором запускаются на выполнение потоки для котят.

В классе *Cat* нужно реализовать еще один метод, в котором вначале на консоль выводится «имя_кошки родила два котенка» (с указанием клички кошки), используйте метод *getName*( )), затем вызывается метод *initAllKitten*( ), а потом на консоль выводится «имя_кошки собрала назад котят в корзинку. Все ее котята в корзинке.».

В папке *theme14* создайте класс *Exercise*. В методе *main*( ) класса *Exercise* определите, сколько будет кошек, их клички. Должно быть не менее двух кошек.

Задание 15.5. В папке *theme14* создайте класс *Exercise* с переменной *public static volatile boolean isStopped*. Инициализируйте ее значением *false*.

В папке *theme14* создайте класс *Clock* (часы), наследуемый от *Thread*. В конструкторе без параметров этого класса установите максимально возможный приоритет для соответствующего потока и запустите его на выполнение.

Реализуйте следующую логику приватного метода *printTikTak*( ) *throws InterruptedException* класса *Clock*: через первые полсекунды

должно выводиться на консоль «Тик», а через вторые полсекунды должно выводиться на консоль «Так».

В классе *Clock* нужно реализовать еще один метод, в котором, пока часы не остановлены, вызывается метод *printTikTak*( ). Для проверки используйте переменную *isStopped*.

В методе *main*( ) класса *Exercise* создайте и запустите часы. Через 2 с их остановите и выведите в консоль «Часы остановлены».

Задание 15.6. В папке *theme14* создайте класс *Exercise* с переменной *public static volatile boolean isStopped*. Инициализируйте ее значением *false*.

В папке *theme14* создайте класс *Clock* (часы), наследуемый от *Thread*. Класс содержит переменные *String cityName, int hours, int minutes, int seconds*. Укажите правильный модификатор доступа для этих переменных.

В конструкторе с четырьмя параметрами класса *Clock* переменные инициализируются, запускается на выполнение соответствующий поток.

В классе *Clock* реализуйте приватный метод *void printTime*( ) *throws InterruptedException* так, чтобы каждую секунду выдавалось время, начиная с установленного в конструкторе. Пример вывода следующий:

В г. Лондон сейчас 23:59:58!

В г. Лондон сейчас 23:59:59!

В г. Лондон сейчас полночь!

В г. Лондон сейчас 0:0:1!

В классе *Clock* нужно реализовать еще один метод, в котором, пока часы не остановлены, вызывается метод *printTime*( ). Для проверки используйте переменную *isStopped*.

В методе *main*( ) класса *Exercise* создайте и запустите часы. Через 4 с остановите их и выведите в консоль «Часы остановлены».

Задание 15.7. В папке *theme14* создайте класс *Exercise* с переменной *public static volatile List<String> list*. Инициализируйте ее списком из пяти элементов.

В статическом блоке класса *Exercise* в список *list* занесите пять строк, каждая из которых начинается словом «Строка», а затем указывается ее порядковый номер.

В папке *theme14* создайте класс *Countdown* (обратный отсчет), реализующий интерфейс от *Runnable*. Класс содержит одну переменную

countFrom, которая инициализируется в конструкторе с одним параметром.

В классе *Countdown* реализуйте логику метода *printCountdown*( ) *throws InterruptedException* так, чтобы каждые полсекунды выводился на консоль объект из переменной *list* в обратном порядке от индекса *countFrom* до нуля. Если, например, был передан индекс 3, то вывод на консоль будет следующий:

Строка 2
Строка 1
Строка 0

В классе *Countdown* нужно реализовать еще один метод, в котором будет вызываться метод *printCountdown*( ).

В методе *main*( ) класса *Exercise* создайте один объект класса *Countdown* и запустите на выполнение соответствующий поток.

Задание 15.8. В папке *theme14* создайте класс *Runway* (взлетная полоса) с приватной переменной типа *Thread*. Создайте для этой переменной геттер, но с именем *getTakingOffPlane*( ), сеттер с именем *setTakingOffPlane*(*Thread t*). Сеттер должен быть синхронизирован по объекту.

В папке *theme14* создайте класс *Airport* (аэропорт) с переменной *public static volatile Runway RUNWAY = new Runway*( ). Предполагается, что в аэропорту одна взлетная полоса.

В папке *theme14* создайте класс *Plane* (самолет), наследуемый от *Thread*. В классе должен быть создан конструктор с одним параметром (названием самолета), вызывающий конструктор суперкласса с этим же параметром. Затем соответствующий поток запускается на выполнение.

В классе *Plane* реализуйте метод *takingOff*( ) (взлетать). Взлет должен занимать 1 с (соответствующий поток засыпает на 1 с).

В классе *Plane* реализуйте метод *waiting*( ) по аналогии с методом *takingOff*( ). Время ожидания не должно превышать время взлета.

В классе *Plane* нужно реализовать еще один метод, в котором если самолет еще не взлетел (*isAlreadyTakenOff = = false*) и если взлетная полоса свободна, то этот самолет выезжает на взлетную полосу (на консоль выводится «Самолет взлетает» с указанием названия самолета), самолет взлетает (на консоль выводится «Самолет уже в небе» с указанием названия самолета, переменная *isAlreadyTakenOff* получает значение *true*), освобождается взлетная полоса.

Если взлетная полоса занята самолетом, то на консоль выводится «Самолет ожидает» с указанием названия самолета, самолет ждет.

В методе *main*( ) класса *Airport* создайте три самолета, например, авиакомпаний Belavia, Lufthansa, Aeroflot.

Задание 15.9. В папке *theme14* создайте класс *Stopwatch* (секундомер), наследуемый от *Thread*. В классе должна быть приватная переменная *seconds* (секунды) типа *int*.

В папке *theme14* создайте класс *Exercise* с методом *main*( ). В методе создайте и запустите секундомер. Затем считайте строку с консоли. Остановите секундомер.

В классе *Stopwatch* реализуйте метод, с помощью которого можно подсчитать количество секунд, прошедших от запуска секундомера до ввода строки. Это количество нужно вывести на консоль.

Задание 15.10. В папке *theme14* создайте класс *TestThread*, реализующий интерфейс *Runnable*.

В классе *TestThread* реализуйте метод *ourInterruptMethod*( ), с помощью которого можно будет сделать так, чтобы поток сам завершился. Нельзя использовать метод *interrupt*( ). Подумайте, какой должен быть возвращаемый тип у метода.

В классе *TestThread* реализуйте еще один метод, в котором в бесконечном цикле, пока поток не прерван, выводится на консоль «Хе-хе», а затем поток засыпает на полсекунды.

В папке *theme14* создайте класс *Exercise* с методом *main*( ). В методе создайте и запустите на выполнение один дочерний поток, используя класс *TestThread*. Вызовите метод *sleep*(3 000) для главного потока. Остановите дочерний поток.

Задание 15.11. В папке *theme14* создайте класс *CountDownRunnable*, реализующий интерфейс *Runnable*. В классе объявите следующие приватные переменные: *countIndexDown* типа *int* и *t* типа *Thread*. Инициализируйте переменную *countIndexDown* некоторым значением, например, 5.

В классе *CountDownRunnable* создайте конструктор с один параметром (именем потока). В конструкторе создайте поток с этим именем и запустите его на выполнение.

В классе *CountDownRunnable* переопределите метод *toString*( ). Он должен возвращать строку, состоящую из имени потока и значения переменной *countIndexDown*, разделенных двоеточием. Используйте метод *getName*( ) для получения имени потока.

В классе *CountDownRunnable* реализуйте еще один метод, в котором пока значение переменной *countIndexDown* не достигнет нуля,

нужно выводить на консоль имя потока и текущее значение переменной *countIndexDown*, затем уменьшить ее значение на один и приостановить выполнение потока на 1 с. Метод не выбрасывает никаких исключений. В случае возникновения исключения выведите на консоль «Поток прерван» с указанием имени потока.

В папке *theme14* создайте класс *CountDownDemo* с методом *main*( ). В методе создайте два потока типа *CountDownRunnable* с разными именами.

Задание 15.12. В папке *theme14* создайте класс *CountUpRunnable*, реализующий интерфейс *Runnable*. В классе объявите следующие приватные переменные: *countIndexUp* типа *int* и *t* типа *Thread*. Переменной *countIndexUp* присвойте значение 1.

В классе *CountUpRunnable* создайте конструктор с один параметром (именем потока). В конструкторе создайте поток с этим именем и запустите его на выполнение.

В классе *CountUpRunnable* переопределите метод *toString*( ). Он должен возвращать строку, состоящую из имени потока и значения переменной *countIndexUp*, разделенных двоеточием. Используйте метод *getName*( ) для получения имени потока.

В классе *CountUpRunnable* реализуйте еще один метод, в котором пока значение переменной *countIndexUp* не достигнет 5, нужно выводить на консоль имя потока и текущее значение переменной *countIndexUp*, затем увеличить ее значение на 1 и приостановить выполнение потока на 1 с. Метод не выбрасывает никаких исключений. В случае возникновения исключения выведите на консоль «Поток прерван» с указанием имени потока.

В папке *theme14* создайте класс *CountUpDemo* с методом *main*( ). В методе создайте два потока типа *CountUpRunnable* с разными именами.

Задание 15.13. В папке *theme14* создайте классы *TestThreads, Accum, ThreadOne, ThreadTwo*.

Разместите по классам строки кода, представленные на рисунке 15. В код классов нужно добавить недостающие закрывающие фигурные скобки. Вывод на консоль в результате выполнения программы должен быть следующим:

Один 98 098
Два 98 099

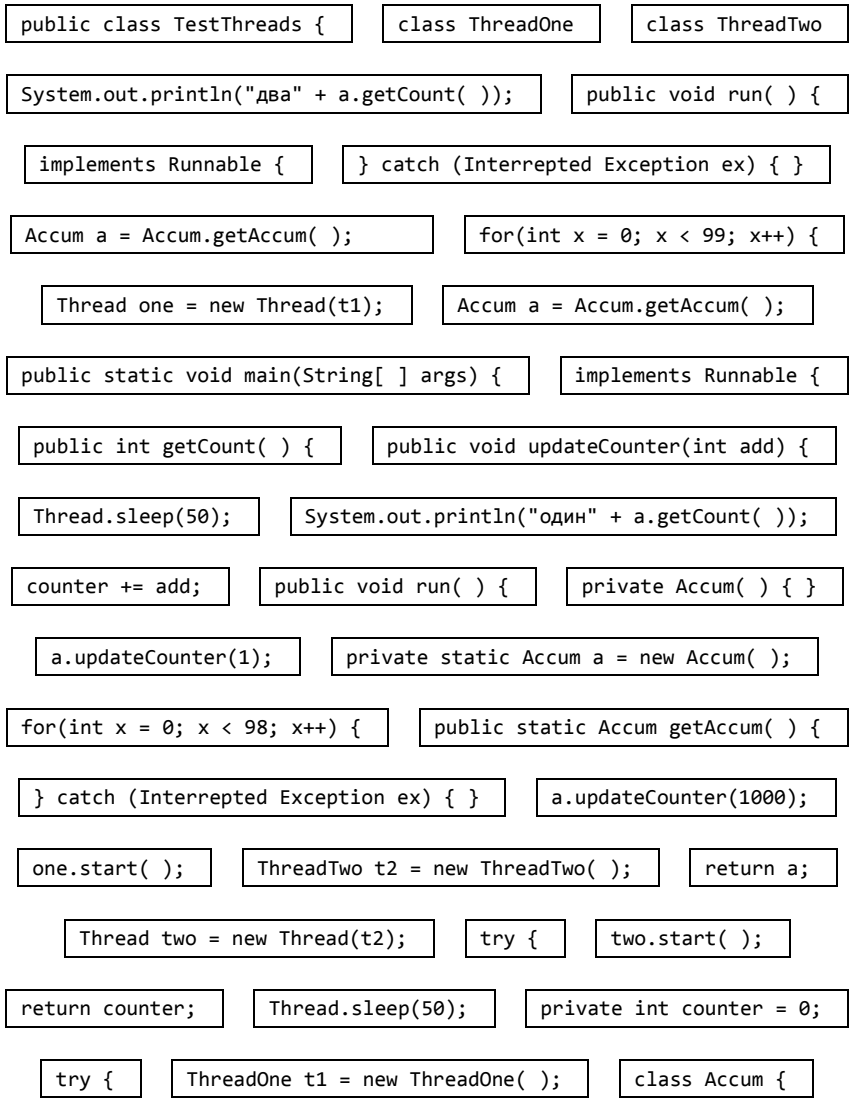

Рисунок 15 – **Части программы к заданию 15.13**

Тема 16. СОЗДАНИЕ ПРОГРАММ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ СРЕДСТВАМИ SWING

16.1. Введение в библиотеку Swing

В Java для создания графического пользовательского интерфейса (ГПИ) обычно используются библиотеки AWT, Swing и JavaFX. Исторически первой и базовой была библиотека AWT (Abstract Window Toolkit). Библиотека AWT содержит базовый набор компонентов, поддерживающих создание вполне работоспособных, но ограниченных по своим возможностям графических пользовательских интерфейсов. Ограниченность библиотеки AWT объясняется, в частности, тем, что ее различные визуальные компоненты транслируются в соответствующие платформенно-зависимые эквиваленты, так называемые равноправные компоненты (peers). Отсюда следует, что внешний вид компонентов AWT определяется не средствами Java, а платформой. Поскольку в компонентах AWT используются ресурсы в виде машинно-зависимого кода, их называют тяжеловесными (heavyweight).

Библиотека компонентов Swing появилась в 1997 г. Swing устраняет ограничения, присущие компонентам AWT, благодаря использованию следующих основных средств: легковесных компонентов и подключаемых стилей оформления. Несмотря на то что программисту почти не приходится использовать эти средства напрямую, именно они составляют фундамент философии проектирования, заложенной в Swing, в значительной мере обусловливают возможности и удобство использования этой библиотеки. Необходимо рассмотреть каждое из них в отдельности. За небольшим исключением все компоненты Swing являются легковесными. Это означает, что они написаны полностью на Java и не зависят от конкретной платформы, поскольку не опираются на платформенно-зависимые равноправные компоненты. Легковесные компоненты обладают рядом существенных преимуществ, к числу которых относятся эффективность и гибкость. Например, легковесный компонент может быть прозрачным, а его форма может отличаться от прямоугольной. Легковесные компоненты не транслируются в платформенно-зависимые равноправные компоненты, поэтому их внешний вид определяет библиотека Swing, а не базовая операционная система. Следовательно, элементы пользовательского интерфейса, созданные средствами Swing, выглядят одинаково на разных платформах. Благодаря тому что каждый компонент Swing визуализируется кодом Java, а не платформенно-зависимыми равноправными компонентами, становится возможным раздельное

управление внешним видом компонента и логикой его функционирования, именно эту задачу решает Swing. Такое разделение предоставляет следующее значительное преимущество: оно позволяет изменить внешний вид компонента, не затрагивая другие его свойства. Иными словами, появляется возможность «подключать» новый стиль оформления к компоненту, не создавая никаких побочных эффектов в коде, использующем данный компонент.

Несмотря на то, что библиотека Swing снимает некоторые ограничения, присущие библиотеке AWT, она не заменяет ее. Напротив, библиотека Swing построена на основе библиотеки AWT. Именно поэтому библиотека AWT до сих пор является важной составной частью Java. Кроме того, в библиотеке Swing применяется тот же самый механизм обработки событий, что и в библиотеке AWT. Классы библиотек AWT и Swing, используемых при создании ГПИ, представлены на рисунке 16.

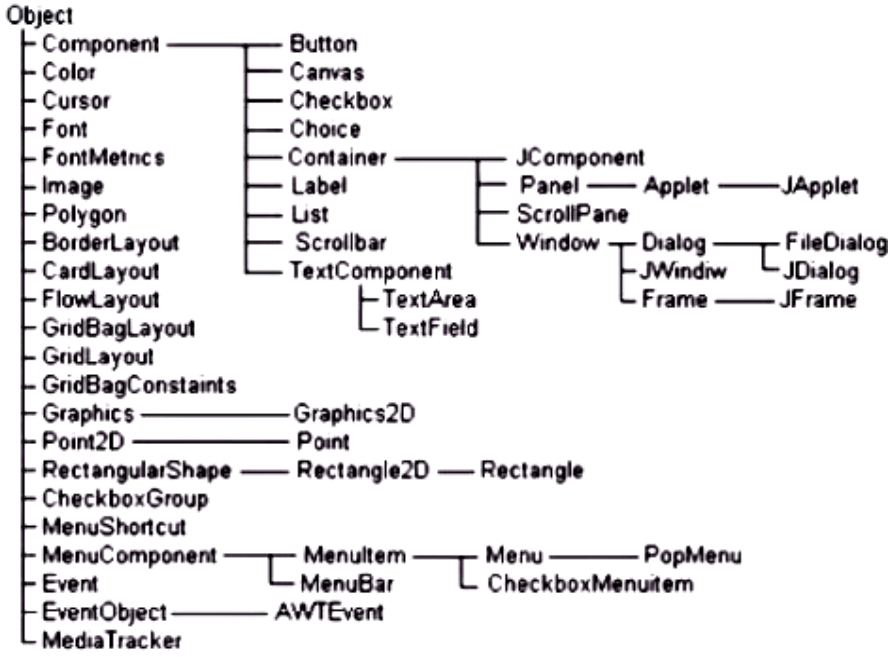

Рисунок 16 – **Классы библиотек AWT и Swing, используемых при создании ГПИ**

ГПИ, создаваемый средствами Swing, состоит из следующих основных элементов: компонентов и контейнеров. Это концептуальное разделение, поскольку все контейнеры также являются компонентами. Отличие этих двух элементов заключается в их следующем назначении: компонент является независимым визуальным элементом управления вроде кнопки или ползунка, а контейнер содержит группу компонентов. Таким образом, контейнер является особым типом компонента и предназначен для хранения других компонентов. Более того, компонент должен находиться в контейнере, чтобы его можно было отобразить. Во всех ГПИ, создаваемых средствами Swing, имеется как минимум один контейнер. Поскольку контейнеры являются компонентами, то один контейнер может содержать другие контейнеры. Благодаря этому в библиотеке Swing можно определить иерархию вложенности, на вершине которой должен находиться контейнер верхнего уровня.

Компоненты библиотеки Swing происходят от класса *JComponent*. Исключением из этого правила являются четыре контейнера верхнего уровня. В классе *JComponent* предоставляются функциональные возможности, общие для всех компонентов. В классе *JComponent* поддерживается подключаемый стиль оформления. Класс *JComponent* наследует классы *Container* и *Component* из библиотеки AWT. Компонент библиотеки Swing построен на основе компонента библиотеки AWT и совместим с ним. Все компоненты Swing представлены классами, определенными в пакете *javax.swing*. Все классы компонентов начинаются с буквы J. Например, класс для создания метки называется *JLabel*, класс для создания кнопки – *JButton*, класс для создания ползунка – *JScrollBar*. Компоненты библиотеки Swing представлены на рисунке 17.

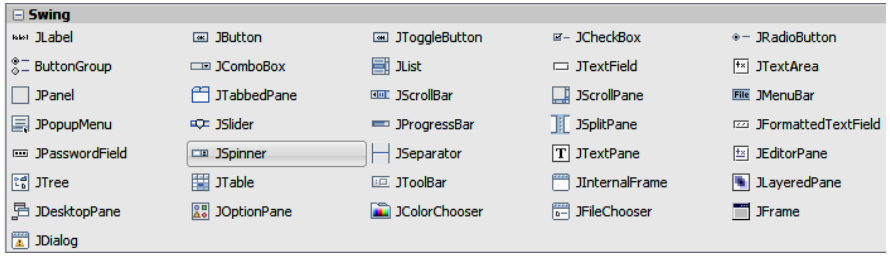

Рисунок 17 – **Компоненты библиотеки Swing**

В библиотеке Swing определены два типа контейнеров. К первому типу относятся контейнеры верхнего уровня, представленные классами *JFrame, JApplet, JWindow* и *JDialog*. Классы этих контейнеров не наследуют от класса *JComponent*, но они наследуют от классов *Component* и *Container* из библиотеки AWT. В отличие от остальных компонентов Swing, которые являются легковесными, компоненты верхнего уровня являются тяжеловесными. Поэтому в библиотеке Swing контейнеры являются особым случаем компонентов. Контейнер верхнего уровня должен находиться на вершине иерархии контейнеров. Контейнер верхнего уровня не содержится ни в одном из других контейнеров. Каждая иерархия вложенности должна начинаться с контейнера верхнего уровня. Таким контейнером в прикладных программах чаще всего является класс *JFrame*, а в апплетах - класс *JApplet*.

Ко второму типу контейнеров, поддерживаемых в библиотеке Swing, относятся легковесные контейнеры. Они наследуют от класса *JComponent*. Примером легковесного контейнера служит класс *JPanel*, который представляет контейнер общего назначения. Легковесные контейнеры нередко применяются для организации и управления группами связанных вместе компонентов, поскольку легковесный контейнер может находиться в другом контейнере. Следовательно, легковесные контейнеры вроде класса *JPanel* можно применять для создания подгрупп связанных вместе элементов управления, содержащихся во внешнем контейнере.

Каждый контейнер верхнего уровня определяет ряд панелей. На вершине иерархии панелей находится корневая панель в виде экземпляра класса *JRootPane*. Класс *JRootPane* представляет легковесный контейнер, предназначенный для управления остальными панелями. Он также помогает управлять дополнительной, хотя и не обязательной строкой меню. Панели, составляющие корневую панель, называются прозрачной панелью, панелью содержимого и многослойной панелью соответственно (рисунок 18).

Прозрачная панель является панелью верхнего уровня. Она находится над всеми панелями и покрывает их полностью. По умолчанию эта панель представлена прозрачным экземпляром класса *JPanel*. Прозрачная панель позволяет управлять событиями от мыши, оказывающими влияние на весь контейнер в целом, а не на отдельный элемент управления, или, например, рисовать поверх любого другого компонента. Как правило, обращаться к прозрачной панели непосредственно не требуется, но если она все же понадобится, то ее нетрудно обнаружить там, где она обычно находится.

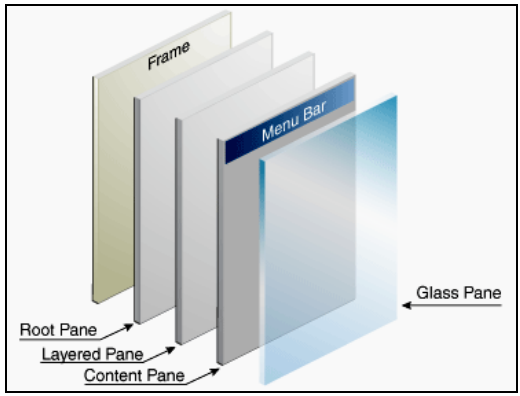

Рисунок 18 – **Панели контейнера верхнего уровня**

Многослойная панель представлена экземпляром класса *JLayeredPane*. Она позволяет задать определенную глубину размещения компонентов. Глубина определяет степень перекрытия компонентов. В связи с этим многослойные панели позволяют задавать упорядоченность компонентов по координате Z, хотя это требуется не всегда. На многослойной панели находится панель содержимого и дополнительно, хотя и не обязательно, строка меню.

Несмотря на то, что прозрачная и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и служат для разных целей, большая часть их возможностей скрыта от пользователей. Прикладная программа чаще всего будет обращаться к панели содержимого, поскольку именно на ней обычно располагаются визуальные компоненты. Когда компонент (например, кнопка) вводится в контейнер верхнего уровня, он оказывается на панели содержимого. По умолчанию панель содержимого представлена непрозрачным экземпляром класса *JPanel*.

16.2. Окно *JFrame*

Каждая GUI-программа запускается в окне, по ходу работы может открывать несколько дополнительных окон.

В библиотеке Swing описан класс *JFrame*, представляющий собой окно с рамкой и строкой заголовка (с кнопками «Свернуть», «Во весь экран» и «Закрыть»). Оно может изменять размеры и перемещаться по экрану.

В Swing есть еще несколько классов окон. Например, *JWindow* – простейшее окно, без рамки и строки заголовка. Обычно с его помощью делается заставка к программе, которая перед запуском должна выполнить несколько продолжительных действий (например, загрузить информацию из базы данных).

Конструктор *JFrame*( ) без параметров создает пустое окно. Конструктор *JFrame*(*String title*) создает пустое окно с заголовком *title*.

Чтобы написать простейшую программу, выводящую на экран пустое окно, потребуются следующие методы:

• *setSize*(*int width, int height*) устанавливает размеры окна. Если не задать размеры, окно будет иметь нулевую высоту независимо от того, что в нем находится, пользователю после запуска придется растягивать окно вручную. Размеры окна включают не только «рабочую» область, но границы и строку заголовка.

• *setDefaultCloseOperation*(*int operation*) позволяет указать действие, которое необходимо выполнить, когда пользователь закрывает окно нажатием на крестик. Обычно в программе есть одно или несколько окон, при закрытии которых программа прекращает работу. Для того, чтобы запрограммировать это поведение, следует в качестве параметра *operation* передать константу EXIT_ON_CLOSE, описанную в классе *JFrame*.

• *setVisible*(*boolean visible*). Когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается данный метод с параметром *true*. Если вызвать его с параметром *false*, окно снова станет невидимым.

Теперь можно написать программу, которая создает окно, выводит его на экран и завершает работу после того, как пользователь закрывает окно. Текст программы имеет следующий вид:

```
import javax.swing.*;
public class MyClass {
   public static void main (String[ ] args) {
      JFrame myWindow = new JFrame("Пробное окно");
      myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      myWindow.setSize(400, 300);
      myWindow.setVisible(true);
   }
}
```

Для работы с большинством классов библиотеки Swing понадобится импортировать пакет *java.swing*.

Перед отображением окна необходимо совершить гораздо больше действий, чем в этой простой программе. Необходимо создать множество элементов управления, настроить их внешний вид, разместить в нужных местах окна. Кроме того, в программе может быть много окон и настраивать их все в методе *main*( ) неудобно и неправильно, поскольку нарушается следующий принцип инкапсуляции: держать вместе данные и команды, которые их обрабатывают. Логичнее было бы, чтобы каждое окно занималось своими размерами и содержимым самостоятельно. Классическая структура программы с окнами в файле *SimpleWindow.java* выглядит следующим образом:

```
public class SimpleWindow extends JFrame {
   SimpleWindow(){
      super("Пробное окно");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(250, 100);
   }
}
```

Классическая структура программы с окнами в файле *Program.java* выглядит следующим образом:

```
public class Program {
   public static void main (String[ ] args) {
      JFrame myWindow = new SimpleWindow();
      myWindow.setVisible(true);
   }
}
```

Из примера видно, что окно описывается в отдельном классе, являющемся наследником *JFrame*. Окно настраивает свой внешний вид и поведение в конструкторе (первой командой вызывается конструктор суперкласса). Метод *main*( ) содержится в другом классе, ответственном за управление ходом программы. Каждый из этих классов очень прост, каждый занимается своим делом, поэтому в них легко разбираться и легко сопровождать (т. е. совершенствовать при необходимости).

Метод *setVisible*( ) не вызывается в классе *SimpleWindow*. Это вполне логично, за тем, где какая кнопка расположена и какие размеры оно должно иметь, следит само окно, а вот принятие решения о том, какое окно в какой момент выводится на экран, является прерогативой управляющего класса программы.

16.3. Панель содержимого

Напрямую в окне элементы управления не размещаются. Для этого служит панель содержимого, занимающая все пространство окна.

Обратиться к этой панели можно методом *getContentPane*( ) класса *JFrame*. С помощью метода *add*(*Component component*) можно добавить на нее любой элемент управления.

Кнопка описывается классом *JButton* и создается конструктором с параметром типа *String* (надписью).

Кнопку можно добавить в панель содержимого окна следующими командами:

```
JButton newButton = new JButton();
getContentPane().add(newButton);
```

В результате получится окно с кнопкой. Кнопка занимает всю доступную площадь окна. Такой эффект полезен не во всех программах, поэтому необходимо изучить различные способы расположения элементов на панели.

16.4. Класс *Container* (контейнер)

Элементы, которые содержат другие элементы, называются контейнерами. Все они являются потомками класса *Container* и наследуют от него следующие полезные методы:

- *Add*(*Component component*) добавляет в контейнер элемент *component*.
- *Remove*(*Component component*) удаляет из контейнера элемент *component*.
- *RemoveAll*( ) удаляет все элементы контейнера.
- *GetComponentCount*( ) возвращает число элементов контейнера.

Кроме перечисленных в классе *Container* определено около двух десятков методов для управления набором компонентов, содержащихся в контейнере. Они похожи на методы класса-коллекции. Контейнер является коллекцией, но коллекцией особого рода (визуальной). Кроме хранения элементов контейнер занимается их пространственным расположением и прорисовкой. В частности, он имеет метод *getComponentAt*(*int x, int y*), возвращающий компонент, в который попадает точка с заданными координатами (координаты отсчитываются от левого верхнего угла компонента) и ряд других.

16.5. Класс *JPanel* (панель)

Панель *JPanel* – это элемент управления, представляющий собой прямоугольное пространство, на котором можно размещать другие элементы. Элементы добавляются и удаляются методами, унаследованными от класса *Container*.

В примере с кнопкой добавленная на панель содержимого кнопка заняла все ее пространство. Это происходит не всегда. У каждой панели есть менеджер размещения, который определяет стратегию взаимного расположения элементов, добавляемых на панель. Его можно изменить методом *setLayout*(*LayoutManager manager*). Чтобы передать в этот метод нужный параметр, необходимо знать, какими бывают менеджеры.

16.6. Менеджеры компоновки

Менеджер компоновки управляет размещением компонентов в контейнере. В Java определено несколько таких менеджеров. Большинство из них входит в состав AWT (т. е. в пакете *java.awt*), но Swing предоставляет также ряд дополнительных менеджеров компоновки. Все менеджеры компоновки являются экземплярами классов, реализующих интерфейс *LayoutManager*. Некоторые из менеджеров компоновки реализуют интерфейс *LayoutManager2*. В таблице 18 перечислен ряд менеджеров компоновки, доступных для разработчиков, использующих библиотеку Swing.

Таблица 18 – **Некоторые менеджеры компоновки библиотеки Swing**

Менеджер	Описание работы
FlowLayout	Простой менеджер компоновки, размещающий компоненты слева направо и сверху вниз. Для некоторых региональных настроек компоненты располагаются справа налево
BorderLayout	Располагает компоненты по центру или по краям контейнера. Этот менеджер принимается по умолчанию для панели содержимого
GridLayout	Располагает компоненты в ячейках сетки как в таблице
GridBagLayout	Располагает компоненты разных размеров в ячейках сетки с регулируемыми размерами
BoxLayout	Располагает компоненты в вертикальном и горизонтальном направлении
SpringLayout	Располагает компоненты с учетом ряда ограничений

16.7. Менеджер последовательного размещения *FlowLayout*

Самый простой менеджер размещения – *FlowLayout*. Он размещает добавляемые на панель компоненты строго по очереди, строка за строкой, в зависимости от размеров панели. Как только очередной элемент не помещается в текущей строке, он переносится на следующую. Это можно наблюдать на следующем примере. Изменим конструктор класса *SimpleWindow* следующим образом:

```
SimpleWindow(){
   super("Пробное окно");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   JPanel panel = new JPanel();
   panel.setLayout(new FlowLayout());
   panel.add(new JButton("Кнопка"));
   panel.add(new JButton("+"));
   panel.add(new JButton("-"));
   panel.add(new JButton("Кнопка с длинной надписью"));
   setContentPane(panel); setSize(250, 100);
}
```

Окно с менеджером размещения *FlowLayout* представлено на рисунке 19.

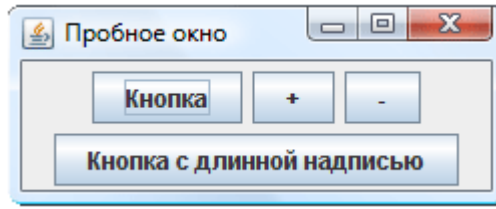

Рисунок 19 – **Окно с менеджером размещения *FlowLayout***

16.8. Менеджер граничного размещения *BorderLayout*

Менеджер размещения *BorderLayout* разделяет панель на пять областей (центральную, верхнюю, нижнюю, правую и левую). В каждую из этих областей можно добавить ровно по одному компоненту,

причем компонент будет занимать всю отведенную для него область. Компоненты, добавленные в верхнюю и нижнюю области, будут растянуты по ширине, добавленные в правую и левую – по высоте, а компонент, добавленный в центр, будет растянут так, чтобы полностью заполнить оставшееся пространство панели.

При добавлении элемента на панель с менеджером размещения *BorderLayout* необходимо дополнительно указывать в методе *add*( ), какая из областей имеется в виду. Для этого служат строки с названиями сторон света: “North”, “South”, “East”, “West” и “Center”. Вместо них рекомендуется использовать следующие константы, определенные в классе *BorderLayout*: NORTH, SOUTH, EAST, WEST и CENTER (поскольку в строке можно допустить ошибку и не заметить этого, а при попытке написать неправильно имя константы компилятор выдаст предупреждение). Если же использовать метод *add*( ) как обычно, с одним параметром, элемент будет добавлен в центр.

Панель содержимого имеет именно такое расположение, поэтому кнопка занимала все окно целиком (она была добавлена в центральную область). Чтобы пронаблюдать эффект *BorderLayout*, необходимо добавить кнопки во все пять областей:

```
   SimpleWindow(){
      super("Пробное окно");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      getContentPane().add(new   JButton("Кнопка"),   Border-
Layout.NORTH);
      getContentPane().add(new JButton("+"), BorderLayout.EAST);
      getContentPane().add(new JButton("-"), BorderLayout.WEST);
      getContentPane().add(new JButton("Кнопка с длинной над-
писью"), BorderLayout.SOUTH);
      getContentPane().add(new JButton("В ЦЕНТР!"));
      setSize(250, 100);
   }
```

Окно с менеджером размещения *BorderLayout* представлено на рисунке 20.

Данное размещение не случайно используется в панели содержимого по умолчанию. Большинство программ пользуются областями по краям окна, чтобы расположить в них панели инструментов, строку состояния и т. п. Ограничение на один компонент в центральной области абсолютно не существенно, ведь этим компонентом может

быть другая панель со множеством элементов и любым менеджером расположения.

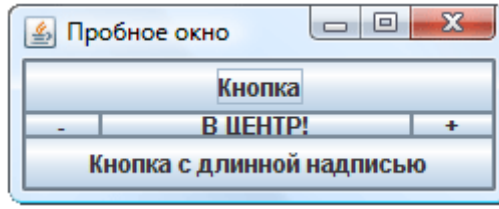

Рисунок 20 – **Окно с менеджером размещения *BorderLayout***

16.9. Менеджер табличного размещения *GridLayout*

GridLayout разбивает панель на ячейки одинаковой ширины и высоты (таким образом, окно становится похожим на таблицу). Каждый элемент, добавляемый на панель с таким расположением, целиком занимает одну ячейку. Ячейки заполняются элементами по очереди, начиная с левой верхней.

Этот менеджер, в отличие от рассмотренных ранее, создается конструктором с параметрами (четыре целых числа). Необходимо указать количество столбцов, строк и расстояние между ячейками по горизонтали и вертикали:

```
SimpleWindow(){
   super("Пробное окно");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   JPanel panel = new JPanel();
   panel.setLayout(new GridLayout(2,3,5,10));
   panel.add(new JButton("Кнопка"));
   panel.add(new JButton("+"));
   panel.add(new JButton("-"));
   panel.add(new JButton("Кнопка с длинной надписью"));
   panel.add(new JButton("еще кнопка"));
   setContentPane(panel); setSize(250, 100);
}
```

Окно с менеджером размещения *GridLayout* представлено на рисунке 21.

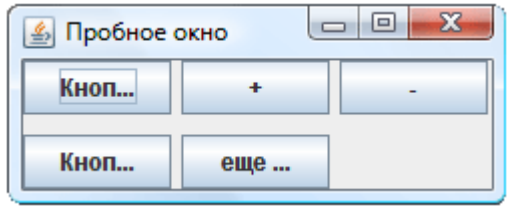

Рисунок 21 – **Окно с менеджером размещения *GridLayout***

16.10. Менеджер блочного размещения *BoxLayout* и класс *Box*

Менеджер *BoxLayout* размещает элементы на панели в строку или в столбец.

Обычно для работы с этим менеджером используют вспомогательный класс *Box*, представляющий собой панель, для которой уже настроено блочное размещение. Создается такая панель не конструктором, а одним из следующих статических методов, определенных в классе *Box*: *createHorizontalBox*( ) и *createVerticalBox*( ).

Элементы, добавленные на панель с блочным размещением, выстраиваются один за другим. Расстояние между элементами по умолчанию является нулевым. Однако вместо компонента можно добавить невидимую «распорку», единственной задачей которой является раздвижение соседних элементов с обеспечением между ними заданного расстояния. Горизонтальная распорка создается статическим методом *createHorizontalStrut*(*int width*), а вертикальная – методом *createVerticalStrut*(*int height*). Оба метода определены в классе *Box*, а целочисленный параметр в каждом из них определяет размер «распорки».

На такую панель можно добавить еще один специальный элемент – своеобразную «пружину». Если размер панели будет больше, чем необходимо для оптимального размещения всех элементов, те из них, которые способны растягиваться, будут стараться заполнить дополнительное пространство собой. Если же разместить среди элементов одну или несколько «пружин», дополнительное свободное пространство будет распределяться и в эти промежутки между элементами. Горизонтальная и вертикальная пружины создаются соответственно методами *createHorizontalGlue*( ) и *createVerticalGlue*( ).

Понять особенности работы этого менеджера лучше на наглядном примере. Расположим четыре кнопки вертикально, поставив между

двумя центральными «пружину», а между остальными – распорки в 10 пикселей:

```
SimpleWindow(){
super("Пробное окно");
setDefaultCloseOperation(EXIT_ON_CLOSE);
Box box = Box.createVerticalBox();
box.add(new JButton("Кнопка"));
box.add(Box.createVerticalStrut(10));
box.add(new JButton("+"));
box.add(Box.createVerticalGlue());
box.add(new JButton("-"));
box.add(Box.createVerticalStrut(10));
box.add(new JButton("Кнопка с длинной надписью"));
setContentPane(box); setSize(250, 100);
}
```

Окно с менеджером размещения *BoxLayout* представлено на рисунке 22.

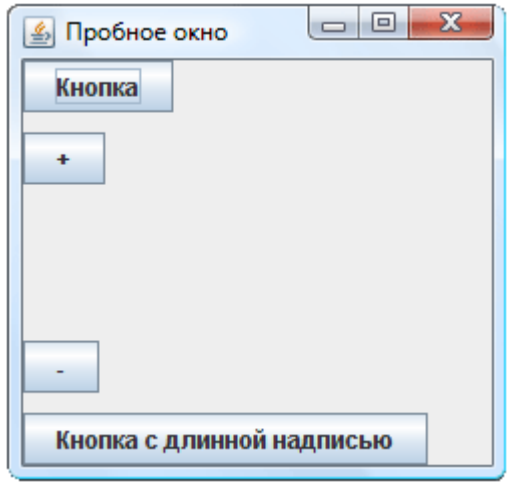

Рисунок 22 – **Окно с менеджером размещения *BoxLayout***

16.11. Ручное размещение элементов

Если в качестве менеджера размещения панели установить *null*, элементы не будут расставляться автоматически. Координаты каждого элемента необходимо в этом случае указать явно, при этом они никак не зависят от размеров панели и координат других элементов. По умолчанию координаты равны нулю (т. е. элемент расположен в левом верхнем углу панели). Размер элемента также необходимо задавать явно (в противном случае его ширина и высота будут равны нулю и элемент отображаться не будет).

Координаты элемента можно задать одним из следующих методов:

```
setLocation(int x, int y),
setLocation(Point point)
```

Эти методы работают аналогично, устанавливая левый верхний угол элемента в точку с заданными координатами. Отличие заключается в способе задания точки. Можно представить точку двумя целыми числами или объектом класса *Point*. Класс *Point* представляет собой ту же пару чисел, его конструктор имеет вид *Point*(*int x, int y*). Получить доступ к отдельной координате можно методами *getX*( ) и *getY*( ).

Размер элемента задается одним из следующих методов:

```
setSize(int width, int height),
setSize(Dimension size)
```

Эти методы работают одинаково. Отличие заключается в способе передачи параметра. Класс *Dimension*, аналогично классу *Point*, просто хранит два числа, имеет конструктор с двумя параметрами *Dimension*(*int width, int height*), позволяет получить доступ к своим составляющим (ширине и высоте) с помощью простых методов *getWidth*( ) и *getHeigth*( ). Для того, чтобы получить текущий размер элемента, можно воспользоваться методом *getSize*( ), возвращающим объект класса *Dimension*.

Создадим панель, с которой не будет связано никакого менеджера размещения, вручную разместим на ней две кнопки:

```
SimpleWindow(){
   super("Пробное окно");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
    JPanel panel = new JPanel();
    panel.setLayout(null);
    JButton button = new JButton("Кнопка");
    button.setSize(80, 30);
    button.setLocation(20,20);
    panel.add(button);
    button = new JButton("Кнопка с длинной надписью");
    button.setSize(120, 40);
    button.setLocation(70,50);
    panel.add(button);
    setContentPane(panel);
    setSize(250, 150);
  }
```

Используем одну и ту же переменную *button* для обращения к обеим кнопкам (причем, второй раз ее описывать не нужно). После осуществления всех необходимых операций с первой кнопкой можно использовать «освободившуюся» переменную для манипуляций со второй.

16.12. Автоматическое определение размеров компонентов

Если у панели есть любой менеджер размещения, она игнорирует явно заданные размеры и координаты всех своих элементов. В этом легко убедиться, заменив в предыдущем примере команду *panel.setLayout*(*null*) на *panel.setLayout*(*new FlowLayout*( )). Менеджер размещения сам определяет координаты и размеры всех элементов.

Способ определения координат элементов очевидным образом вытекает из алгоритмов работы каждого менеджера.

В некоторых случаях компоненты стараются заполнить все доступное им пространство. Например, всю центральную область в случае менеджера *BorderLayout* или всю ячейку в менеджере *GridLayout*. В панели с менеджером *FlowLayout*, напротив, элементы никогда не пытаются выйти за определенные границы.

Каждый визуальный компонент имеет три типа размеров: минимально допустимый, максимально допустимый и предпочтительный. Узнать, чему равны эти размеры для данного компонента можно с помощью следующих методов:

```
getMinimumSize(),
```

```
getPreferredSize(),
getMaximumSize().
```

Методы возвращают результат типа *Dimension*. Они запрограммированы в соответствующем классе. Например, у кнопки минимальный размер – нулевой, максимальный размер не ограничен, а предпочтительный зависит от надписи на кнопке (вычисляется как размер текста надписи плюс размеры полей).

Менеджер *FlowLayout* всегда устанавливает предпочтительные размеры элементов. Менеджер *BorderLayout* устанавливает предпочтительную ширину правого и левого, а также предпочтительную высоту верхнего и нижнего. Остальные размеры подстраиваются под доступное пространство панели. Менеджер *GridLayout* пытается подогнать размеры всех элементов под размер ячеек. Менеджер *BoxLayout* ориентируется на предпочтительные размеры.

Когда элемент старается занять все доступное ему пространство, он «учитывает» пожелания не быть меньше своих минимальных или больше максимальных размеров.

Всеми тремя размерами можно управлять с помощью следующих методов *set*:

```
setMinimumSize(Dimension size),
setPreferredSize(Dimension size),
setMaximumSize(Dimension size).
```

Чаще всего используется простой прием, когда элементу «не рекомендуется» увеличиваться или уменьшаться относительно своих предпочтительных размеров. Это легко сделать следующей командой:

```
element.setMinimumSize(element.getPreferredSize());
```

16.13. «Упаковка» окна

В рассмотренных выше примерах размер окна явно задавался методом *setSize*( ). Когда используется какой-либо менеджер расположения, расставляющий элементы и изменяющий их размеры по собственным правилам, трудно сказать заранее, какие размеры окна будут самыми подходящими.

Безусловно, наиболее подходящим будет вариант, при котором все элементы окна имеют предпочтительные размеры или близкие к ним.

Если вместо явного указания размеров окна вызвать метод *pack*( ), они будут подобраны оптимальным образом с учетом предпочтений всех элементов, размещенных в этом окне.

Оцените работу этого метода, заменив в каждом из вышеприведенных примеров команду `setSize(250, 100);` на команду `pack();`.

Когда панель не имеет метода размещения, эта команда не работает (поскольку панель не имеет алгоритма для вычисления своего предпочтительного размера).

16.14. Класс *JComponent*

Все визуальные компоненты библиотеки Swing унаследованы от класса *JComponent*. Сам этот класс является абстрактным и непосредственно не используется, но все визуальные компоненты наследуют его методы.

setEnabled(*boolean enabled*) используется для управления активностью компонента. При вызове этого метода с параметром *false* компонент переходит в неактивное состояние. Для каждого наследника *JComponent* эта «неактивность» может быть переопределена по-разному. Например, неактивная кнопка не нажимается, не реагирует на наводящуюся мышь и отображается монохромным серым цветом.

isEnabled( ) возвращает *true*, если элемент активен, в противоположном случае – *false*.

setVisible(*boolean visible*) управляет видимостью компонента. Он уже использовался для отображения окна *JFrame*. Большинство элементов управления, в отличие от окна, по умолчанию являются видимыми (поэтому данный метод не был вызван после создания кнопок в предыдущих примерах).

isVisible( ) возвращает *false*, если элемент невидим, в противоположном случае – *true*.

С помощью метода *setBackground*(*Color color*) можно изменить цвет заднего фона компонента. Однако эффект будет иметь место лишь в том случае, если компонент непрозрачен (некоторые компоненты, например, метка *JLabel* по умолчанию являются прозрачными).

Непрозрачность устанавливается методом *setOpaque*(*boolean opaque*) с параметром *true*.

Методы *getBackground*( ) и *isOpaque*( ) возвращают текущий цвет заднего фона и непрозрачность компонента.

16.15. Метка *JLabel*

В большинстве визуальных библиотек метка – один из самых простейших компонентов. Она представляет собой обычный текст, который выводится в заданном месте окна и используется для вывода вспомогательной текстовой информации (подписи к другим элементам, инструкции и предупреждения для пользователя). В Swing метка позволяет достичь более интересных эффектов. Во-первых, помимо текста можно использовать значок. Во-вторых, с ее помощью можно выводить отформатированный текст.

Текст и значок метки можно задать в ее конструкторе. У нее есть несколько конструкторов с различными параметрами, в частности следующие:

- *JLabel*(*String text*), который создает метку с надписью *text*.
- *JLabel*(*Icon image*), который создает метку со значком *image*.
- *JLabel*(*String text, Icon image, int align*), который создает метку с надписью *text* и значком *image*. Третий параметр задает выравнивание текста вместе со значком. В качестве него может быть использована одна из констант, описанных в интерфейсе *SwingConstants* (LEFT, RIGHT, CENTER).

Для примера необходимо создать окно с меткой, созданной при помощи третьего конструктора. Будут использованы два класса, один из которых назовем *SimpleWindow* и унаследуем его от класса окна *JFrame*. В его конструкторе будут создаваться и размещаться все элементы окна. Второй класс будет создавать это окно и отображать его на экране (код будет таким же, как и в предыдущих примерах).

Напишем в конструкторе класса *SimpleWindow* следующий код:

```
SimpleWindow(){
   super("Окно с надписью");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   JLabel label = new JLabel("Метка со значком и с надписью",
      new ImageIcon("1.gif"), SwingConstants.RIGHT);
   getContentPane().add(label); pack();
}
```

Окно с меткой представлено на рисунке 23.

Чтобы убедиться, что выравнивание по правому краю работает, необходимо немного растянуть окно, чтобы ширина метки стала более оптимальной.

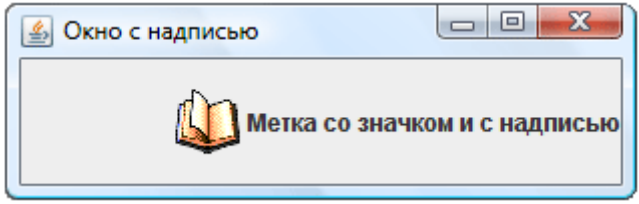

Рисунок 23 – **Окно с меткой**

В библиотеке Swing метка (и не только она) может быть настроена для отображения отформатированного текста в формате HTML. Для этого необходимо, чтобы строка, устанавливаемая в качестве надписи метки, начиналась с тега <html>. После этого можно использовать в строке любые теги языка HTML версии 3.2, они будут преобразовываться в соответствующие атрибуты форматирования. В этом легко убедиться, изменив в предыдущем примере строку с вызовом конструктора следующим образом:

```
JLabel label = new JLabel("<html>К этой метке применено "
+ "HTML-форматирование, включая: <ul><li> <i>курсив</i>,"
+ "<li><b>полужирный</b> <li><font size = +2> увеличение
размера </font>"
+ "<li>маркированный список </ul>");
```

Окно с меткой с текстом в формате HTML представлено на рисунке 24.

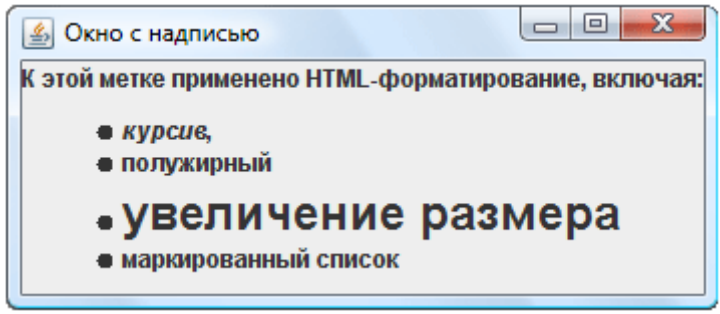

Рисунок 24 – **Окно с меткой с текстом в формате HTML**

Поскольку текст нашей надписи достаточно длинный, строка для удобства восприятия разбивается на части и используется оператор конкатенации.

Основные методы класса *JLabel* следующие:

- *getText*( ), который возвращает текущий текст надписи метки;
- *setText*(*String text*), который задает новый текст надписи;
- *getIcon*( ), который возвращает значок метки;
- *setIcon*(*Icon image*), который устанавливает новый значок; в качестве значка обычно используется объект простого класса *ImageIcon*.

getVerticalAlignment( ), *setVerticalAlignment*(*int align*), *getHorizontalAlignment*( ), *setHorizontalAlignment*(*int align*) – методы, которые позволяют получить текущее или установить новое выравнивание (по горизонтали и вертикали) метки относительно ее границ. Возможные положения описаны в интерфейсе *SwingConstants*.

getVerticalTextPosition( ), *setVerticalTextPosition*(*int align*), *getHorizontalTextPosition*( ), *setHorizontalTextPosition*(*int align*) – методы, которые позволяют получить текущее или установить новое выравнивание текста относительно значка. Возможные положения описаны в интерфейсе *SwingConstants*.

getIconTextGap( ), *setIconTextGap*(*int gap*) позволяет получить или задать расстояние между текстом и значком метки в пикселях.

16.16. Кнопка *JButton*

Кнопка – это прямоугольник с текстом (и (или) значком), по которому пользователь щелкает, когда хочет выполнить какое-то действие (или о чем-то сигнализировать).

Кнопка создается одним из пяти конструкторов, в частности *JButton*( ), *JButton*(*String text*), *JButton*(*Icon icon*), *JButton*(*String text, Icon icon*), параметры которых говорят сами за себя.

Кроме обычного значка можно назначить кнопке еще несколько (для различных состояний). Метод *setRolloverIcon*(*Icon icon*) позволяет задать значок, который будет появляться при наведении на кнопку мыши, *setPressedIcon*(*Icon icon*) – значок для кнопки в нажатом состоянии, *setDisableIcon*(*Icon icon*) – значок для неактивной кнопки. Каждому из этих методов соответствует метод *get*.

Метод *setMargin*(*Insets margin*) позволяет задать величину отступов от текста надписи на кнопке до ее полей. Объект класса *Insets*, который передается в этот метод, может быть создан конструктором с

четырьмя целочисленными параметрами, задающими величину отступов: *Insets*(*int top, int left, int bottom, int right*). Метод *getMargin*( ) возвращает величину текущих отступов в виде объекта того же класса.

Все методы класса *JLabel*, описанные ранее, присутствуют и в классе *JButton*. С помощью этих методов можно изменять значок и текст надписи на кнопке, а также управлять их взаимным расположением относительно друг друга и относительно края кнопки (с учетом отступов).

Посредством методов *setBorderPainted*(*boolean borderPainted*), *setFocusPainted*(*boolean focusPainted*), *setContentAreaFilled*(*boolean contentAreaFilled*) можно отключать (параметром *false*) и включать обратно (параметром *true*) прорисовку рамки, прорисовку фокуса (кнопка, на которой находится фокус, выделяется пунктирным прямоугольником) и закраску кнопки в нажатом состоянии.

Для примера создадим следующую кнопку со значком и надписью, изменим ее отступы и расположение текста относительно значка (текст будет выровнен влево и вверх относительно значка). Текст программы имеет следующий вид:

```
   SimpleWindow(){
      super("Окно с кнопкой");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JButton button = new JButton("Кнопка", new ImageI-
con("1.gif"));
      button.setMargin(new Insets(0, 10, 20, 30));
      button.setVerticalTextPosition(SwingConstants.TOP);
      button.setHorizontalTextPosition(SwingConstants.LEFT);
      getContentPane().add(button);
      pack();
   }
```

Кнопка со значком и надписью представлена на рисунке 25.

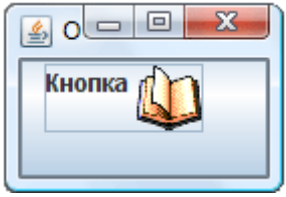

Рисунок 25 – **Кнопка со значком и надписью**

16.17. Компоненты *JToggleButton, JCheckBox, JRadioButton*

Компонент *JToggleButton* представляет собой кнопку, которая может находиться в двух состояниях: нажатом и отпущенном. Когда пользователь щелкает мышкой по такой кнопке, она изменяет свое состояние. Именно таким образом ведут себя кнопки форматирования на инструментальной панели текстового редактора. Кнопка [I] не только устанавливает или убирает курсивное начертание в выделенном тексте, но и сигнализирует о его наличии или отсутствии.

Основной конструктор – *JToggleButton*(*String text, Icon icon, boolean selected*) создает кнопку с заданными надписью, значком и текущим состоянием. Кнопку можно перевести в требуемое состояние программным путем, вызвав метод *setSelected*(*boolean selected*). Метод *isSelected*( ) возвращает *true*, если кнопка выбрана (т. е. находится в нажатом состоянии), в противоположном случае – *false*.

От класса *JToggleButton* унаследован класс *JCheckBox* – флажок. Этот класс имеет точно такой же набор конструкторов и методов, т. е. не расширяет функциональность родительского класса. Единственное различие между ними заключается во внешнем виде: *JCheckBox* выглядит не как кнопка, а как небольшой квадратик, в котором можно поставить или убрать галочку.

Аналогичным образом ведет себя класс *JRadioButton* – переключатель или радиокнопка, внешне выглядящая как пустой кружок, когда она не выделена и кружок с точкой в выделенном состоянии.

Несмотря на то, что классы *JCheckBox* и *JRadioButton* ведут себя абсолютно одинаково (аналогично их общему предку *JToggleButton*), их принято использовать в различных ситуациях. В частности, *JRadioButton* предполагает выбор единственной альтернативы из нескольких возможных, несколько таких объектов объединяются в одну группу (чаще всего эта группа визуально обозначается рамкой) и при выборе одного из элементов группы предыдущий выбранный элемент переходит в состояние «не выбран».

Для того чтобы получить такое поведение, используется специальный контейнер *ButtonGroup* – взаимоисключающая группа (создается конструктором без параметров). Если добавить в один такой контейнер несколько элементов *JRadioButton*, то выбранным всегда будет только один из них.

В *ButtonGroup* могут быть добавлены не только переключатели, но флажки и кнопки выбора (и даже обычные кнопки). При разработке интерфейса необходимо следовать устоявшемуся подходу, согласно кото-

рому во взаимоисключающую группу следует объединять объекты *JRadioButton* (в некоторых случаях *JToggleButton*), но не *JCheckBox*.

Метод *add*(*AbstractButton button*) добавляет элемент в группу. Метод *getElements*( ) возвращает все ее элементы в виде коллекции *Enumeration*. По коллекции можно пройтись итератором и найти выделенный элемент.

Рассмотрим пример, в котором создаются две кнопки выбора, два флажка и два переключателя. Кнопки выбора и переключатели объединены в группы *ButtonGroup*. Для того, чтобы обвести каждую пару элементов рамкой, необходимо расположить каждую пару элементов на отдельной панели:

```
   SimpleWindow(){
      super("Пример с кнопками выбора, флажками и переключа-
телями");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      ImageIcon icon = new ImageIcon("1.gif");
      // будем использовать один значок на все случаи
      Box mainBox = Box.createVerticalBox();
      Box box1 = Box.createVerticalBox();
      JToggleButton tButton1 = new JToggleButton("Кнопка вы-
бора 1");
      JToggleButton tButton2 = new JToggleButton("Кнопка вы-
бора 2", icon);
      ButtonGroup bg = new ButtonGroup();
      // создаем группу взаимного исключения
      bg.add(tButton1);
      bg.add(tButton2);
      // сделали кнопки tButton1 и tButton2 взаимоисключающими
      box1.add(tButton1);
      box1.add(tButton2);
      // добавили кнопки tButton1 и tButton2 на панель box1
      box1.setBorder(new TitledBorder("Кнопки выбора"));
      Box box2 = Box.createVerticalBox();
      JCheckBox check1 = new JCheckBox("Флажок 1");
      JCheckBox check2 = new JCheckBox("Флажок 2", icon);
      box2.add(check1); box2.add(check2);
      // добавили флажки на панель box2
      box2.setBorder(new TitledBorder("Флажки"));
      Box box3 = Box.createVerticalBox();
```

```
        JRadioButton rButton1 = new JRadioButton("Переключатель
1");
        JRadioButton rButton2 = new JRadioButton("Переключатель
2", icon);
        bg = new ButtonGroup();
        // создаем группу взаимного исключения
        bg.add(rButton1);
        bg.add(rButton2);
        // сделали радиокнопки взаимоисключающими
        box3.add(rButton1);
        box3.add(rButton2);
        // добавили радиокнопки на панель box3
        box3.setBorder(new TitledBorder("Переключатели"));
        mainBox.add(box1);
        mainBox.add(box2);
        mainBox.add(box3);
        setContentPane(mainBox);
        pack();
    }
```

Окно с компонентами *JToggleButton*, *JCheckBox*, *JradioButton* представлено на рисунке 26.

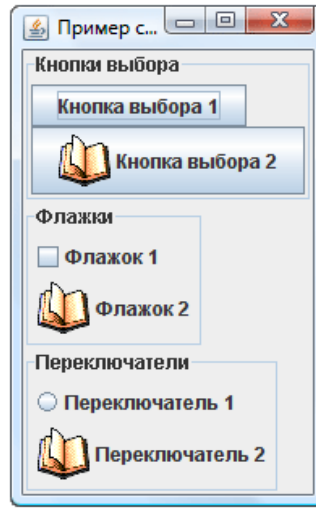

Рисунок 26 – **Окно с компонентами *JToggleButton, JCheckBox, JRadioButton***

Можно наблюдать особенности работы кнопок выбора, флажков и переключателей. У флажков или переключателей рисунок заменяет элемент выделения. Рисунок не показывает, выбран ли данный объект, что может сбить пользователя с толку. Необходимо установить отдельный рисунок для выделенного состояния, что достигается методом *setSelectedIcon*(*Icon icon*). Добавьте в нужные места команду `check2.setSelectedIcon(new ImageIcon("2.gif"));` и команду `rButton2.setSelectedIcon(new ImageIcon("2.gif"));`.

Файл *2.gif*, равно как и файл *1.gif* должен находиться в доступном для программы месте (в директории проекта).

В рассмотренном примере рамки имеют различную ширину. Измените пример таким образом, чтобы все рамки были одинаковы по ширине. Используйте другой менеджер расположения главной панели (вместо *BoxLayout*).

16.18. Текстовое поле *JTextField*

Текстовое поле – простой и часто используемый компонент, предназначенный для ввода небольших по объему (записываемых в одну строку) текстовых данных. Для создания текстового поля чаще всего используются следующие конструкторы:

- *JTextField*(*int columns*), который создает пустое текстовое поле, ширина которого достаточна для размещения указанного количества символов. При этом пользователь может вводить в текстовое поле строку какой угодно длины (она просто будет прокручиваться).
- *JTextField*(*String text*), который создает текстовое поле с начальным текстом.
- *JTextField*(*String text, int columns*), который устанавливает ширину и начальный текст.

Занести текст в поле можно методом *setText*(*String text*). Метод *getText*( ) возвращает содержимое текстового поля целиком. *GetText* (*int offset, int length*) – фрагмент содержимого длины *length*, который начинается с символа *offset*.

Часть текста в поле может выделяться (как программным путем, так и в результате действий пользователя). Метод *getSelectedText*( ) позволяет получить выделенную часть текста. Заменить выделенный текст другим можно с помощью метода *replaceSelection*(*String content*). Методы *getSelectionStart*( ) и *getSelectionEnd*( ) возвращают границы выделенного участка, а методы *setSelectionStart*(*int start*) и *setSelectionEnd*(*int end*) изменяют их.

Метод *getCaretPosition*( ) возвращает позицию курсора (каретки) в текстовом поле, а метод *setCaretPosition*(*int position*) позволяет задать ее программно. Методом *setCaretColor*(*Color color*) можно изменить цвет курсора.

По умолчанию текст в поле прижимается к левому краю. Изменить это можно методом *setHorizontalAlignment*(*int align*), в качестве параметра передается одна из следующих констант выравнивания, определенных в этом же классе *JTextField*: LEFT, CENTER, RIGHT.

16.19. Поле для ввода пароля *JPasswordField*

JPasswordField является прямым потомком *JTextField*, поэтому для него справедливо все сказанное выше. Отличие заключается в том, что весь введенный в него текст скрыт от посторонних глаз. Он заменяется звездочками или другим символом, установить который позволяет метод *setEchoChar*(*char echo*), а получить – *getEchoChar*( ).

Чаще всего *JPasswordField* применяется для ввода пароля. Метод *getText*( ) позволяет получить этот пароль, но пользоваться им не рекомендуется (злоумышленник может проанализировать содержимое оперативной памяти и перехватить этот пароль). Вместо него следует использовать метод *getPassword*( ), возвращающий массив символов *char*[ ]. После того, как введенный пароль будет обработан (например, сравнен с реальным паролем) рекомендуется заполнить этот массив нулями, чтобы следов в оперативной памяти не осталось.

16.20. Область для ввода текста *JTextArea*

JTextArea также является потомком *JTextField* и наследует все его методы. В отличие от текстового поля область для ввода текста позволяет ввести не одну строку, а несколько. В связи с этим *JTextArea* предлагает несколько дополнительных функций. Во-первых, это способность переносить слова на соседнюю строку целиком, которой управляет метод *setWrapStyleWord*(*boolean wrapStyle*). Если вызвать этот метод с параметром *true*, то слова не будут разрываться в том месте, где они «натыкаются» на границу компонента, а будут целиком перенесены на новую строку. Во-вторых, это способность переносить текст (длинные строки будут укладываться в несколько строк вместо одной, уходящей за границы компонента; этой способностью управляет метод *setLineWrap*(*boolean lineWrap*). Методы *isWrapStyle-*

Word( ) и *isLineWrap*( ) возвращают текущее состояние данных способностей (*true* – активирована, *false* – деактивирована).

При создании *JTextArea* чаще всего используют конструктор *JTextArea*(*int rows*, *int columns*), устанавливающий высоту (количество строк) и ширину (количество символов) компонента.

Для работы со своим содержимым *JTextArea* дополнительно предлагает два удобных метода. Метод *append*(*String text*) добавляет строку *text* в конец уже имеющегося текста, а метод *insert*(*String text*, *int position*) вставляет ее в позицию *position*.

Пронаблюдаем эти три компонента на наглядном примере. Создадим простое окно, в котором разместим их с помощью менеджера *BorderLayout*. Текст программы имеет следующий вид:

```
   SimpleWindow(){
      super("Пример текстовых компонентов");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JTextField textField = new JTextField("Текстовое поле",
20);
      textField.setCaretColor(Color.RED);
      textField.setHorizontalAlignment(JTextField.RIGHT);
      JPasswordField passwordField = new JPasswordField(20);
      passwordField.setEchoChar('$');
      passwordField.setText("пароль");
      JTextArea textArea = new JTextArea(5, 20);
      textArea.setLineWrap(true);
      textArea.setWrapStyleWord(true);
      for (int i = 0; i <= 20; i++)
      textArea.append("Область для ввода текстового содержи-
мого");
      getContentPane().add(textField, BorderLayout.NORTH);
      getContentPane().add(textArea);
      getContentPane().add(passwordField, BorderLayout.SOUTH);
      pack();
   }
```

Окно с текстовыми компонентами представлено на рисунке 27.

Для того чтобы лучше понять особенности работы текстовой области, замените по очереди *true* на *false* в вызовах методов *setLineWrap*( ) и *setWrapStyleWord*( ). Пронаблюдайте за изменением работы

компонента. Изменяйте размеры окна, чтобы видеть, каким образом текст перестраивается под доступное ему пространство.

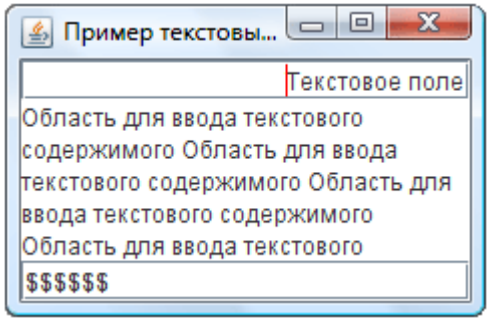

Рисунок 27 – **Окно с текстовыми компонентами**

Расположите в окне те же самые три компонента, но с помощью менеджера размещения *FlowLayout*, который всегда устанавливает для компонентов их предпочтительные размеры. Пронаблюдайте за тем, как ведет себя область *JTextArea* при добавлении в нее нового текста.

16.21. Панель прокрутки *JScrollPane*

Наблюдая за поведением компонента *JTextArea* в предыдущем примере, можно легко обнаружить проблемы, которые возникают, когда тексту становится «тесно» в рамках отведенного места. В зависимости от используемого менеджера расположения текст либо обрезается, уходя за границы компонента, либо раздвигает эти границы (в любом случае остается ограничен размером окна). В таких случаях типично использование полос прокрутки, но в Swing полосы прокрутки сами собой не появляются.

Добавить к компоненту полосы прокрутки на самом деле очень просто. Для этого служит компонент *JScrollPane* – панель прокрутки. Чаще всего она просто «надевается» на требуемый объект посредством собственного конструктора, принимающего этот объект в качестве параметра. Например, чтобы текстовая область *textArea* из предыдущего примера обрела полосы прокрутки, необходимо заменить команду `getContentPane().add(textArea);` на команду `getContentPane().add(new JScrollPane(textArea));`.

В этой команде создается панель с полосами прокрутки, в нее помещается объект *textArea*, сама панель добавляется в панель содержимого окна. Теперь текст свободно прокручивается. В случае применения менеджера *FlowLayout* или *BoxLayout* компонент *JTextArea* не будет подстраиваться под свое содержимое (будет иметь предпочтительный размер, соответствующий параметрам конструктора), при необходимости отображать полоски прокрутки.

Полезными методами *JScrollPane* являются следующие:

• *setHorizontalScrollBarPolicy*(*int policy*), который позволяет задать стратегию работы с горизонтальной полосой прокрутки. Возможные значения представлены константами HORIZONTAL_SCROLLBAR_ ALWAYS (отображать всегда), HORIZONTAL_SCROLLBAR_AS_NEEDED (отображать при необходимости) и HORIZONTAL_SCROLLBAR_NEVER (не отображать никогда). Данные константы определены в интерфейсе *ScrollPaneConstants*.

• *setVerticalScrollBarPolicy*(*int policy*) позволяет задать стратегию работы с вертикальной полосой прокрутки посредством констант VERTICAL_SCROLLBAR_ALWAYS, VERTICAL_SCROLLBAR_ AS_ NEEDED, VERTICAL_SCROLLBAR_NEVER.

16.22. Инструментальная панель *JToolBar*

Большинство программных продуктов предоставляют удобные инструментальные панели, расположенные вдоль границ окна программы и содержащие кнопки, выпадающие списки и другие элементы управления, обычно соответствующие командам меню. В Swing для инструментальных панелей разработан визуальный компонент *JToolBar*, в котором заложена просто потрясающая функциональность.

Создадим окно с менеджером расположения *BorderLayout*, разместим по центру область для ввода текста *JTextArea*, а к верхней границе прикрепим инструментальную панель с тремя кнопками и одним разделителем:

```
SimpleWindow(){
   super("Пример использования JToolBar");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   JTextArea textArea = new JTextArea(5, 20);
   getContentPane().add(textArea);
```

```
    JToolBar toolBar = new JToolBar("Инструментальная па-
нель");
    toolBar.add(new JButton("Кнопка 1"));
    toolBar.add(new JButton("Кнопка 2"));
    toolBar.addSeparator();
    toolBar.add(new JButton("Кнопка 3"));
    getContentPane().add(toolBar, BorderLayout.NORTH);
    pack();
  }
```

Окно с инструментальной панелью представлено на рисунке 28.

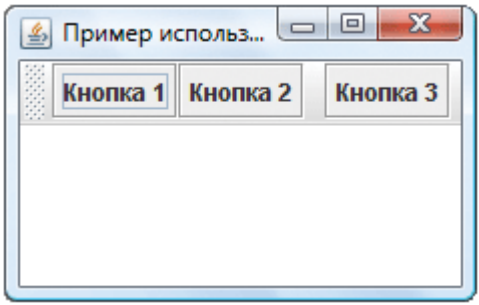

Рисунок 28 – **Окно с инструментальной панелью**

Запустите пример и поэкспериментируйте с инструментальной панелью. Попробуйте отсоединить ее от верхней границы окна и прикрепить к какой-либо другой. Отсоедините ее от границ окна так, чтобы панель стала самостоятельным окном. При этом панель всегда отображается над родительским окном, даже если именно оно, а не панель является активным. Если закрыть самостоятельную панель кнопкой с крестиком, она вернется в свое окно, в то место, где она была закреплена последний раз. Окно со смещенной инструментальной панелью представлено на рисунке 29.

Конструктор *JToolBar*(*String title*) создает горизонтальную панель с заданным заголовком. Горизонтальная панель предназначена для прикрепления к верхней либо нижней границе родительской панели (имеющей расположение *BorderLayout*). Для создания вертикальной панели используется конструктор *JToolBar*(*String title, int orientation*), в котором параметр *orientation* задается константой VERTICAL из

интерфейса *SwingConstants*. Также доступны конструкторы *JToolBar*( ) и *JToolBar*(*int orientation*), создающие панель без заголовка.

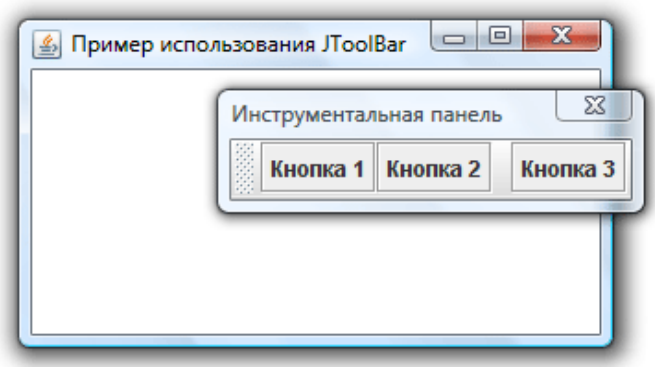

Рисунок 29 – **Окно со смещенной инструментальной панелью**

SetFloatable(*boolean floatable*) разрешает либо запрещает (по умолчанию разрешает) пользователю открeплять панель от места ее начального расположения. Ему соответствует метод *isFloatable*( ) возвращающий *true*, если открeплять панель разрешено.

Add(*Component component*) добавляет на инструментальную панель новый элемент управления. Взаимосвязанные группы элементов управления принято разделять с помощью линии или пустого пространства. Метод *addSeparator*( ) добавляет такой разделитель.

16.23. Выпадающий список *JComboBox*

Выпадающий список – весьма распространенный элемент управления. Он содержит множество вариантов, из которых пользователь может выбрать один либо (если выпадающий список это позволяет) ввести свой собственный.

Создать выпадающий список можно конструктором по умолчанию *JComboBox*( ), после чего добавлять в него элементы методом *addItem*(*Object item*), добавляющим новый элемент в конец списка, или методом *insertItemAt*(*Object item, int index*), позволяющим уточнить позицию, в которую требуется вставить элемент. Однако проще использовать конструктор, в котором сразу указываются все элементы выпадающего списка. Таких конструкторов два (*JComboBox*

(*Object*[ ] *elements*) и *JComboBox*(*Vector elements*). Работают они одинаково, так что использовать массив или вектор – это вопрос удобства разработчика.

Чаще всего в выпадающий список добавляют строки, но, как это следует из сигнатур описанных выше методов, он может содержать вообще любые объекты. Любой объект преобразуется к строке методом *toString*( ), именно эта строка и будет представлять его в выпадающем списке.

Метод *getItemAt*(*int index*) позволяет обратиться к произвольному элементу.

Метод *removeAllItems*( ) удаляет из *JComboBox* все элементы, а метод *removeItem*(*Object item*) – конкретный элемент (при условии, что он содержался в списке).

Метод *getSelectedIndex*( ) позволяет получить индекс выбранного пользователем элемента (элементы нумеруются начиная с нуля), а метод *getSelectedItem*( ) возвращает сам выбранный объект. Сделать конкретный элемент выбранным можно и программно, воспользовавшись методом *setSelectedIndex*(*int index*) или *setSelectedItem*(*Object item*).

Чтобы пользователь мог ввести свой вариант, который не присутствует в списке, должен быть вызван метод *setEditable*(*boolean editable*) с параметром *true*. Ему соответствует метод *isEditable*( ).

Рассмотрим пример, в котором создается выпадающий список из трех элементов и выбирается второй. Строка, представляющая третий элемент, использует HTML-теги. HTML-теги работают не только в метках:

```
SimpleWindow(){
   super("Пример использования JComboBox");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   String[] elements = new String[] {"Вася",
      "Петя",
      "<html><font size = +1 color = yellow>Иван</font>"};
   JComboBox combo = new JComboBox(elements);
   combo.setSelectedIndex(1);
   JPanel panel = new JPanel();
   panel.add(combo);
   setContentPane(panel);
   setSize(200,200);
}
```

Окно с выпадающим списком представлено на рисунке 30.

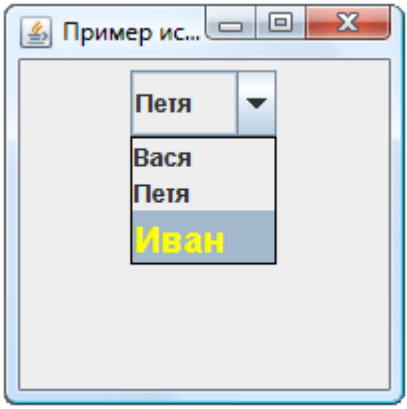

Рисунок 30 – **Окно с выпадающим списком**

16.24. Ползунок *JSlider*

Ползунок позволяет пользователю выбрать некоторое число из диапазона доступных значений, наглядно представив этот диапазон. Против наглядности у ползунка есть один недостаток, он занимает достаточно много места.

Основным конструктором ползунка является *JSlider*(*int orientation, int min, int max, int value*). Первый параметр – ориентация ползунка (HORIZONTAL или VERTICAL). Остальные параметры указывают соответственно минимальное, максимальное и текущее значение. Изменить эти значения позволяют методы *setOrientation*(*int*), *setMinimum*(*int min*), *setMaximum*(*int max*), *setValue*(*int value*), а получить текущие – соответствующие им методы *get*. Чаще всего используется метод *getValue*( ), чтобы определить, какое значение выбрал при помощи ползунка пользователь.

Шкала ползунка может быть украшена делениями. Метод *setMajorTickSpacing*(*int spacing*) позволяет задать расстояние, через которое будут выводиться большие деления, а метод *setMinorTickSpacing*(*int spacing*) – расстояние, через которые будут выводиться маленькие деления. Метод *setPaintTicks*(*boolean paint*) включает или отключает прорисовку этих делений. Метод *setSnapToTicks*(*boolean snap*) включает или отключает «прилипание» ползунка к делениям. Если вызвать

этот метод с параметром *true*, пользователь сможет выбрать при помощи ползунка только значения, соответствующие делениям. Метод *setPaintLabels*(*boolean paint*) включает или отключает прорисовку меток под большими делениями.

Пример использования перечисленных методов следующий:

```
   SimpleWindow(){
      super("Пример использования JSlider");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JSlider slider = new JSlider(JSlider.HORIZONTAL, 50,
150, 70);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      slider.setPaintTicks(true);
      slider.setPaintLabels(true);
      slider.setSnapToTicks(true);
      JPanel panel = new JPanel();
      panel.add(slider);
      setContentPane(panel);
      pack();
   }
```

Окно с ползунком представлено на рисунке 31.

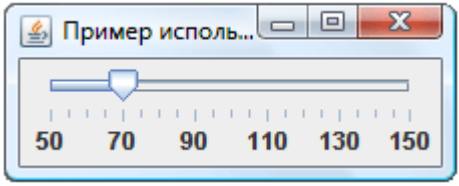

Рисунок 31 – **Окно с ползунком**

16.25. Панель со вкладками *JTabbedPane*

Многим программам бывает необходимо разместить в одном окне большое количество элементов управления, некоторые из которых

(списки, деревья, текстовые области и т. д.) могут к тому же занимать приличное пространство. Чаще всего такая необходимость возникает, когда для работы программе необходимо множество входных данных. Для того, чтобы не вводить дополнительных окон и не перегружать интерфейс, часто используется панель с закладками. Ее можно воспринимать как множество страниц (вкладок), каждая из которых занимает все доступное пространство, за исключением полосы с краю (это может быть любой край), где отображаются ярлычки с названиями страниц. Когда пользователь щелкает по ярлычку, открывается соответствующая ему страница. На каждой странице размещено несколько элементов управления (как правило, они группируются по смыслу).

Создать панель со вкладками можно простым конструктором, в котором определяется только месторасположение ярлычков (LEFT, RIGHT, TOP или BOTTOM). Иногда бывает полезен конструктор *JTabbedPane*(*int orientation, int layout*), в котором второй параметр принимает значения, соответствующие константам SCROLL_TAB_LAYOUT (если все ярлычки не помещаются, появляется полоса прокрутки) или WRAP_TAB_LAYOUT (ярлычки могут располагаться в несколько рядов).

После этого можно добавлять вкладки методом *addTab*( ), имеющим несколько вариантов. В частности, метод *addTab*(*String title, Component tab*) добавляет закладку с указанием текста ярлычка, а метод *addTab*(*String title, Icon icon, Component tab*) позволяет задать также и значок к ярлычку. В качестве вкладки обычно служит панель с размещенными на ней элементами управления.

Создадим панель с десятью вкладками, на каждой из которых поместим по кнопке. Все эти вкладки создадим в цикле *for*, чтобы не писать много кода. Текст программы имеет следующий вид:

```
SimpleWindow(){
   super("Пример использования JTabbedPane");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.TOP,
      JTabbedPane.WRAP_TAB_LAYOUT);
   for (int i = 1; i <= 10; i++) {
      JPanel panel = new JPanel();
      panel.add(new JButton("Кнопка № " + i));
      tabbedPane.add("Панель " + i, panel);
   }
```

```
        getContentPane().add(tabbedPane);
        setSize(300,200);
    }
```

Окно со вкладками представлено на рисунке 32.

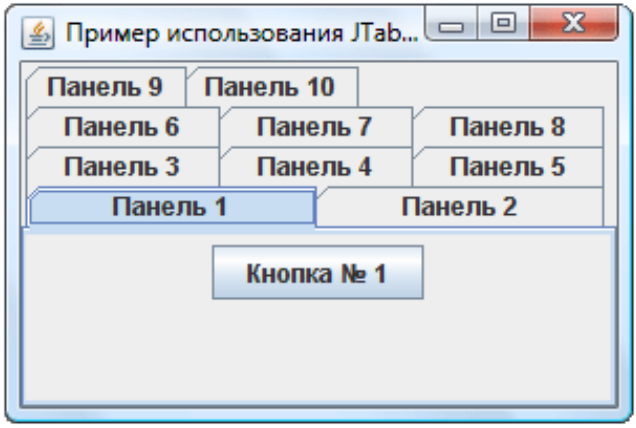

Рисунок 32 – **Окно со вкладками**

16.26. Список *JList*

Список содержит группу элементов, аналогично выпадающему списку *JComboBox*, но обладает двумя отличительными особенностями. Во-первых, на экране видны одновременно несколько элементов списка. Во-вторых, пользователь может выбрать в списке не один элемент, а несколько (если установлен соответствующий режим выделения).

Создать список можно с помощью конструктора, работающего на основе массива *Object*[ ] или вектора *Vector* (аналогично *JComboBox*). Метод *setVisibleRowCount*(*int count*) устанавливает количество видимых элементов списка. Остальные элементы будут уходить за его пределы или прокручиваться, если поместить список в *JScrollPane* (это рекомендуется сделать).

По умолчанию пользователь может выбрать в списке любое число элементов, держа нажатой клавишу Ctrl. Это можно изменить, вызвав

метод *setSelectionMode*(*int mode*), в котором параметр задается одной из следующих констант класса *ListSelectionModel*:

- SINGLE_SELECTION (может быть выделен только один элемент);
- SINGLE_INTERVAL_SELECTION (может быть выделено несколько элементов, но составляющих непрерывный интервал);
- MULTIPLE_INTERVAL_SELECTION (может быть выделено произвольное количество смежных и несмежных элементов).

Выделенный элемент списка (если он один) можно получить методом *getSelectedValue*( ). Если таких несколько, метод вернет первый из них. Метод *getSelectedValues*( ) возвращает все выделенные элементы списка в виде массива *Object*[ ]. Аналогично работают методы *getSelectedIndex*( ) и *getSelectedIndices*( ), только возвращают они не сами выделенные элементы, а их индексы. Всем этим методам соответствуют методы *set*, так что выделить элементы списка можно и программно.

Пример, который иллюстрирует некоторые из этих возможностей *JList*, имеет следующий вид:

```
   SimpleWindow(){
      super("Пример с JList");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      Object[] elements = new Object[] {"Колбаса",
         "<html><font color = red>Масло",
            "Сгущенное молоко"};
      JList list = new JList(elements);
      list.setVisibleRowCount(5);
      list.setSelectionMode(ListSelectionModel.SINGLE_INTERVA
L_SELECTION);
      list.setSelectedIndices(new int[] {1,2});
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(new JScrollPane(list));
      setSize(200,150);
   }
```

Окно со списком *JList* представлено на рисунке 33.

Для того, чтобы эффективно добавлять и удалять элементы из списка, управлять их отображением, добавлять и удалять выделенные элементы поодиночке, необходимо ознакомиться с моделью данных списка.

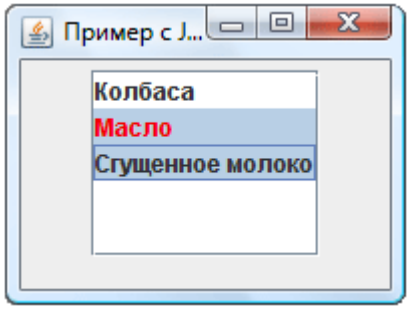

Рисунок 33 – **Окно со списком *JList***

16.27. Окно входа в систему

В качестве реального примера законченного окна необходимо рассмотреть окно для входа в систему. Это окно содержит два текстовых поля для ввода логина и пароля, подписи к этим полям и кнопки *OK* и *Отмена*.

В нашем окне нет элементов, которые имеет смысл растягивать на все доступное пространство, поэтому будет использован один из менеджеров, сохраняющих предпочтительные размеры компонентов (*BoxLayout*). Проще всего представить окно как три горизонтальные панели, объединенные в одной вертикальной. В первой из них будет надпись «Логин:» и текстовое поле. Во второй – надпись «Пароль:» и поле для ввода пароля. В третьей будут размещены две кнопки. При этом необходимо учесть следующее:

- Кнопки *OK* и *Отмена* принято прижимать к правому краю окна, поэтому в начале третьей горизонтальной панели необходимо добавить «пружину».
- Должно выдерживаться аккуратное расстояние между элементами. Для стиля Java разработаны следующие рекомендации: тесно связанные элементы (такие как текстовое поле и подпись к нему) должны находиться на расстоянии друг от друга на 6 пикселей; логически сгруппированные элементы – на 12 пикселей (две верхние панели и пара кнопок). Все остальные элементы должны находиться на расстоянии 17 пикселей друг от друга. Не следует забывать и про рамку окна.
- Элементы должны быть аккуратно выровнены. Надписи, которые необходимо поместить перед текстовыми полями, наверняка бу-

дут разной длины, из-за этого поля будут сдвинуты относительно друг друга, что не рекомендуется. Поэтому следует принудительно задать у надписей одинаковую ширину.

• При увеличении размеров окна текстовые поля будут неэстетично изменять свою высоту. Можно ее зафиксировать, а можно просто запретить окну изменять свои размеры после упаковки командой *setResizable*(*false*). Окно входа в систему представлено на рисунке 34.

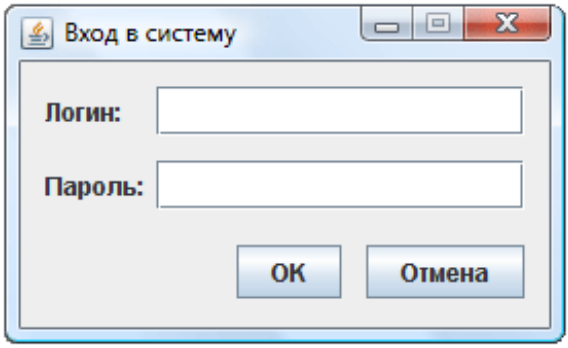

Рисунок 34 – **Окно входа в систему**

Итоговый код окна имеет следующий вид:

```
   public class LoginWindow extends JFrame {
   /* Для того, чтобы впоследствии обращаться к содержимому текстовых полей, рекомендуется сделать их членами класса окна */
      JTextField loginField;
      JPasswordField passwordField;
      LoginWindow(){
         super("Вход в систему");
         setDefaultCloseOperation(EXIT_ON_CLOSE);
         // Настраиваем первую горизонтальную панель (для ввода логина)
         Box box1 = Box.createHorizontalBox();
         JLabel loginLabel = new JLabel("Логин:");
         loginField = new JTextField(15);
         box1.add(loginLabel);
```

```
        box1.add(Box.createHorizontalStrut(6));
        box1.add(loginField);
        // Настраиваем вторую горизонтальную панель (для
ввода пароля)
        Box box2 = Box.createHorizontalBox();
        JLabel passwordLabel = new JLabel("Пароль:");
        passwordField = new JPasswordField(15);
        box2.add(passwordLabel);
        box2.add(Box.createHorizontalStrut(6));
        box2.add(passwordField); //
        Настраиваем третью горизонтальную панель (с кнопками)
        Box box3 = Box.createHorizontalBox();
        JButton ok = new JButton("OK");
        JButton cancel = new JButton("Отмена");
        box3.add(Box.createHorizontalGlue());
        box3.add(ok);
        box3.add(Box.createHorizontalStrut(12));
        box3.add(cancel);
        // Уточняем размеры компонентов
        loginLa-
bel.setPreferredSize(passwordLabel.getPreferredSize());
        // Размещаем три горизонтальные панели на одной вер-
тикальной
        Box mainBox = Box.createVerticalBox();
        mainBox.setBorder(new EmptyBorder(12,12,12,12));
        mainBox.add(box1);
        mainBox.add(Box.createVerticalStrut(12));
        mainBox.add(box2);
        mainBox.add(Box.createVerticalStrut(17));
        mainBox.add(box3);
        setContentPane(mainBox);
        pack();
        setResizable(false);
    }
  }
```

16.28. Обработка событий в Swing

Графический интерфейс пользователя включает в себя не только расположение в окне необходимых элементов управления, но и назначение реакции на пользовательские события. Большая честь действий в оконных программах выполняется в ответ на выбор пользователем команд меню, нажатие кнопок, а иногда даже просто в ответ на ввод нового символа в текстовое поле.

Современный подход к обработке событий основывается на модели делегирования событий, определяющей стандартные и согласованные механизмы их генерации и обработки. В основе модели лежит довольно простая концепция: источник генерирует событие и уведомляет об этом одного или нескольких слушателей. В этой схеме слушатель просто ожидает получения события.

Событие (event) – это объект, описывающий изменение состояния источника. Событие может быть сгенерировано при взаимодействии пользователя с элементами пользовательского интерфейса, например, в результате щелчка на кнопке, ввода символа с клавиатуры, выбора элемента списка или щелчка мышью.

Источник события – это объект, генерирующий событие. Для того чтобы получать уведомления о конкретном виде события, слушатель должен быть зарегистрирован в его источнике. Для каждого вида события определен собственный метод регистрации. Общая форма объявления методов регистрации имеет следующий вид:

```
public void addТипListener(ТипListener элемент)
```

Тип – это имя события, а элемент – ссылка на слушателя события. Например, метод, регистрирующий слушателя событий от клавиатуры, называется *addKeyListener*( ), а метод, регистрирующий слушателя события перемещения мыши, – *adc*!*MouseMotionListener*( ). При наступлении события все слушатели оповещаются об этом и получают копию объекта данного события. Источник должен также предоставлять метод, позволяющий слушателю отменить регистрацию для определенного вида событий. Общая форма объявления такого метода имеет следующий вид:

```
public void removeТипListener{ТипListener элемент)
```

Тип – это имя события, а элемент – ссылка на слушателя данного события. Например, для удаления слушателя событий от клавиатуры следует вызвать метод *removeKeyListener*( ).

Слушатель события (listener) – это объект, получающий уведомления о наступлении события. К нему предъявляются следующие основные требования:

- он должен быть зарегистрирован в одном или нескольких источниках, чтобы получать от них уведомления о конкретных видах событий;
- он должен реализовать методы для получения и обработки таких уведомлений.

Слушатель события находится в постоянном ожидании, пока в источнике, в котором он зарегистрирован, не наступит соответствующее событие, при возникновении которого слушатель получает управление. Также слушателю передается объект события (источник), чтобы он смог правильно на него отреагировать. Таким образом, источник вызывает метод-обработчик события, определенный в классе, являющемся блоком прослушивания.

В таблице 19 приведены определенные в пакете *java.awt.event* типы событий, соответствующие им слушатели, а также методы, определенные в каждом интерфейсе слушателя.

Таблица 19 – **Классы событий**

Класс события	Интерфейс слушателя	Обработчики события
ActionEvent	ActionListener	actionPerformed(ActionEvent e)
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ComponentEvent	ComponentListener	componentResized(ComponentEvent e)
		componentMoved(ComponentEvent e)
		componentShown(ComponentEvent e)
		componentHidden(ComponentEvent e)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent e)
		componentRemoved(ContainerEvent e)
FocusEvent	FocusListener	focusGained(FocusEvent e)
		focusLost(FocusEvent e)
ItemEvent	ItemListener	itemStateChanged(ItemEvent e)
KeyEvent	KeyListener	keyPressed(KeyEvent e)
		keyReleased(KeyEvent e)
		keyTyped(KeyEvent e)

Окончание таблицы 19

Класс события	Интерфейс слушателя	Обработчики события
MouseEvent	MouseListener	mouseClicked(MouseEvent e)
		mousePressed(MouseEvent e)
		mouseReleased(MouseEvent e)
		mouseEntered(MouseEvent e)
		mouseExited(MouseEvent e)
	MouseMotionListener	mouseDragged(MouseEvent e)
		mouseMoved(MouseEvent e)
TextEvent	TextListener	textValueChanged(TextEvent e)
WindowEvent	WindowListener	windowOpened(WindowEvent e)
		windowClosing(WindowEvent e)
		windowClosed(WindowEvent e)
		windowIconified(WindowEvent e)
		windowDeiconified(WindowEvent e)
		windowActivated(WindowEvent e)

Корнем иерархии классов событий является суперкласс *EventObject* из пакета *java.util*. Данный класс содержит следующие методы: *getSource*( ), возвращающий источник событий; *toString*( ), возвращающий строчный эквивалент события. Чтобы узнать, в каком объекте произошло событие, нужно вызвать метод *getSource*( ), возвращающий значение типа *object*. Следовательно, один и тот же слушатель можно подключить к разным источникам.

16.29. Интерфейс *MouseListener* и обработка событий от мыши

События от мыши – один из самых популярных типов событий. Практически любой элемент управления способен сообщить о том, что на него навели мышь, щелкнули по нему и т. д. Об этом будут оповещены все зарегистрированные слушатели событий от мыши.

Кнопка для входа в систему из предыдущего примера должна реагировать на щелчок по ней, проверяя имя и пароль, введенные пользователем.

Слушатель событий от мыши должен реализовать интерфейс *MouseListener*. В этом интерфейсе перечислены следующие методы:

• *public void mouseClicked*(*MouseEvent event*) (выполнен щелчок мышкой на наблюдаемом объекте);

• *public void mouseEntered*(*MouseEvent event*) (курсор мыши вошел в область наблюдаемого объекта);

• *public void mouseExited*(*MouseEvent event*) (курсор мыши вышел из области наблюдаемого объекта);

• *public void mousePressed*(*MouseEvent event*) (кнопка мыши нажата в момент, когда курсор находится над наблюдаемым объектом);

• *public void mouseReleased*(*MouseEvent event*) (кнопка мыши отпущена в момент, когда курсор находится над наблюдаемым объектом).

Чтобы обработать нажатие на кнопку, требуется описать класс, реализующий интерфейс *MouseListener*, причем метод *mouseClicked*( ) должен содержать обработчик события. Далее необходимо создать объект этого класса и зарегистрировать его как слушателя интересующей кнопки. Для регистрации слушателя используется метод *addMouseListener*(*MouseListener listener*).

Опишем класс слушателя в пределах класса окна *SimpleWindow* после конструктора. Обработчик события будет проверять, ввел ли пользователь логин «Иван» (пароль проверять не будет) и выводить сообщение об успехе или неуспехе входа в систему:

```
   class MouseL implements MouseListener {
      public void mouseClicked(MouseEvent event) {
         if (loginField.getText().equals("Иван"))
            JOptionPane.showMessageDialog(null, "Вход выполнен");
         else  JOptionPane.showMessageDialog(null,  "Вход  НЕ
выполнен");
      }
      public void mouseEntered(MouseEvent event) {}
      public void mouseExited(MouseEvent event) {}
      public void mousePressed(MouseEvent event) {}
      public void mouseReleased(MouseEvent event) {}
   }
```

В вышеприведенном примере слушатель был сделан вложенным классом класса *SimpleWindow*, чтобы он мог легко получить доступ к его внутренним полям *loginField* и *passwordField*. Хотя реально мы обрабатываем только одно из пяти возможных событий мыши, описывать пришлось все пять методов (четыре имеют пустую реализа-

цию). В противном случае класс пришлось бы объявить абстрактным (ведь он унаследовал от интерфейса пустые заголовки методов), мы не смогли бы создать объект этого класса. Мы должны создать объект слушателя и прикрепить его к кнопке. Для этого в код конструктора *SimpleWindow*( ) необходимо добавить следующую команду:

```
ok.addMouseListener(new MouseL());
```

Это можно сделать сразу после следующей команды:

```
JButton ok = new JButton("OK");
```

Чтобы кнопка *ok* обрела слушателя, который будет обрабатывать нажатие на нее, понадобилось описать новый (вложенный) класс. Иногда вместо вложенного класса можно обойтись анонимным. Таким образом, можно заменить вложенный класс анонимным. Для этого описание класса *MouseL* можно просто удалить, а команду `ok.addMouseListener(new MouseL());` заменить на следующее:

```
ok.addMouseListener(new MouseListener() {
   public void mouseClicked(MouseEvent event) {
      if (loginField.getText().equals("Иван"))
         JOptionPane.showMessageDialog(null, "Вход выполнен");
      else  JOptionPane.showMessageDialog(null,  "Вход  НЕ
выполнен");
   }
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
});
```

Программа стала выглядеть загроможденной главным образом из-за того, что помимо полезного метода *mouseClicked*( ) пришлось определять пустые реализации всех остальных, не нужных методов. В принципе, этого можно избежать.

Класс *MouseAdapter* реализует интерфейс *MouseListener*, определяя пустые реализации для каждого из его методов. Можно унаследовать своего слушателя от этого класса и переопределить те методы, которые нужны.

В результате предыдущее описание слушателя будет выглядеть следующим образом:

```
ok.addMouseListener(new MouseAdapter() {
   public void mouseClicked(MouseEvent event) {
      if (loginField.getText().equals("Иван"))
         JOptionPane.showMessageDialog(null, "Вход выполнен");
      else  JOptionPane.showMessageDialog(null,  "Вход  НЕ
выполнен");
   }
});
```

16.30. Слушатель фокуса *FocusListener*

Слушатель *FocusListener* отслеживает моменты, когда объект получает фокус (становится активным) или теряет его. Концепция фокуса очень важна для оконных приложений. В каждый момент времени в окне может быть только один активный (находящийся в фокусе) объект, который получает информацию о нажатых на клавиатуре клавишах (т. е. реагирует на события клавиатуры), о прокрутке колесика мышки и т. д. Пользователь активирует один из элементов управления нажатием мышки или с помощью клавиши *Tab* (переключаясь между ними).

Интерфейс *FocusListener* имеет следующие методы:

- *public void focusGained*(*FocusEvent event*) (вызывается, когда наблюдаемый объект получает фокус);
- *public void focusLost*(*FocusEvent event*) (вызывается, когда наблюдаемый объект теряет фокус).

16.31. Слушатель колесика мышки *MouseWheelListener*

Слушатель *MouseWheelListener* оповещается при вращении колесика мыши в тот момент, когда данный компонент находится в фокусе. Этот интерфейс содержит всего один метод: *public void mouseWheelMoved*(*MouseWheelEvent event*).

16.32. Слушатель клавиатуры *KeyListener*

Слушатель *KeyListener* оповещается, когда пользователь работает с клавиатурой в тот момент, когда данный компонент находится в фокусе. В интерфейсе определены следующие методы:

• *public void keyTyped*(*KeyEvent event*) (вызывается, когда с клавиатуры вводится символ);

• *public void keyPressed*(*KeyEvent event*) (вызывается, когда нажата клавиша клавиатуры);

• *public void keyReleased*(*KeyEvent event*) (вызывается, когда отпущена клавиша клавиатуры).

Аргумент *event* этих методов способен дать весьма ценные сведения. В частности, команда *event.getKeyChar*( ) возвращает символ типа *char*, связанный с нажатой клавишей. Если с нажатой клавишей не связан никакой символ, возвращается константа CHAR_UNDEFINED. Команда *event.getKeyCode*( ) возвратит код нажатой клавиши в виде целого числа типа *int*. Его можно сравнить с одной из многочисленных констант, определенных в классе *KeyEvent* (VK_F1, VK_SHIFT, VK_D, VK_MINUS и т. д.). Методы *isAltDown*( ), *isControlDown*( ), *isShiftDown*( ) позволяют узнать, не была ли одновременно нажата одна из клавиш-модификаторов Alt, Ctrl или Shift.

16.33. Слушатель изменения состояния *ChangeListener*

Слушатель *ChangeListener* реагирует на изменение состояния объекта. Каждый элемент управления по своему определяет понятие «изменение состояния». Например, для панели со вкладками *JTabbedPane* это переход на другую вкладку, для ползунка *JSlider* – изменение его положения, кнопка *JButton* рассматривает как смену состояния щелчок на ней. Таким образом, хотя событие это достаточно общее, необходимо уточнять его специфику для каждого конкретного компонента. В интерфейсе определен следующий метод: *public void stateChanged*(*ChangeEvent event*).

16.34. Слушатель событий окна *WindowListener*

Слушатель *WindowListener* может быть привязан только к окну и оповещается о следующих событиях, произошедших с окном:

• *Public void windowOpened*(*WindowEvent event*) – окно открылось.

• *Public void windowClosing*(*WindowEvent event*) – попытка закрытия окна (например, пользователь нажал на крестик). Слово «попытка» означает, что данный метод вызовется до того, как окно будет закрыто и может воспрепятствовать этому (например, вывести диалог типа «Вы уверены?» и отменить закрытие окна, если пользователь выберет «Нет»).

• *Public void windowClosed*(*WindowEvent event*) – окно закрылось.

• *Public void windowIconified*(*WindowEvent event*) – окно свернуто.

• *Public void windowDeiconified*(*WindowEvent event*) – окно развернуто.

• *Public void windowActivated*(*WindowEvent event*) – окно стало активным.

• *Public void windowDeactivated*(*WindowEvent event*) – окно стало неактивным.

16.35. Слушатель событий компонента *ComponentListener*

Слушатель *ComponentListener* оповещается, когда наблюдаемый визуальный компонент изменяет свое положение, размеры или видимость. В интерфейсе присутствуют следующие методы:

• *Public void componentMoved*(*ComponentEvent event*) вызывается, когда наблюдаемый компонент перемещается (в результате вызова команды *setLocation*( ), работы менеджера размещения или еще по какой-то причине).

• *Public void componentResized*(*ComponentEvent event*) вызывается, когда изменяются размеры наблюдаемого компонента.

• *Public void componentHidden*(*ComponentEvent event*) вызывается, когда компонент становится невидимым.

• *Public void componentShown*(*ComponentEvent event*) вызывается, когда компонент становится видимым.

16.36. Слушатель выбора элемента *ItemListener*

Слушатель *ItemListener* реагирует на изменение состояния одного из элементов, входящих в состав наблюдаемого компонента. Например, выпадающий список *JComboBox* состоит из множества элементов, слушатель реагирует, когда изменяется выбранный элемент. Также данный слушатель оповещается при выборе либо отмене выбора флажка *JCheckBox* или переключателя *JRadioButton*, изменении

состояния кнопки *JToggleButton* и т. д. Слушатель обладает методом *public void itemStateChanged*(*ItemEvent event*).

16.37. Универсальный слушатель *ActionListener*

Среди многочисленных событий, на которые реагирует каждый элемент управления (и о которых он оповещает соответствующих слушателей, если они к нему присоединены), есть одно основное, вытекающее из самой сути компонента и обрабатываемое значительно чаще, чем другие. Например, для кнопки – это щелчок на ней, а для выпадающего списка – выбор нового элемента.

Для отслеживания и обработки такого события может быть использован особый слушатель *ActionListener*, имеющий метод *public void actionPerformed*(*ActionEvent event*).

У использования *ActionListener* есть небольшое преимущество в эффективности (при обработке нажатия на кнопку не надо реагировать на четыре лишних события; ведь даже если методы-обработчики пустые, на вызов этих методов все равно тратятся ресурсы). Очень удобно запомнить и постоянно использовать один класс с одним методом и обращаться к остальным лишь в тех относительно редких случаях, когда возникнет такая необходимость.

Обработка нажатия на кнопку *ok* в нашем примере легко переписывается для *ActionListener* следующим образом:

```
ok.addMouseListener(new ActionListener() {
   public void actionPerformed(ActionEvent event) {
      if (loginField.getText().equals("Иван"))
         JOptionPane.showMessageDialog(null, "Вход выполнен");
      else  JOptionPane.showMessageDialog(null,  "Вход  НЕ
выполнен");
   }
});
```

16.38. Работа с меню в библиотеке Swing

Неотъемлемой частью современных оконных программ является меню, представляющее собой удобно сгруппированный набор команд. Меню бывает двух типов: главное и контекстное. Главное меню располагается вдоль верхней границы окна и содержит команды, относящиеся ко всему приложению (точнее, все команды, которые

можно выполнить, работая с данным окном). Контекстное меню вызывается нажатием правой кнопки мышки на конкретном объекте и содержит команды, которые могут быть применены именно к этому объекту.

16.39. Создание главного меню

Главное меню окна представлено в Swing классом *JMenuBar*. По сути своей этот класс представляет собой панель с менеджером расположения *BoxLayout* (по горизонтали), в которую можно добавлять не только элементы меню, а также все, что угодно (выпадающие списки, панели с закладками). Для удобства пользования программой предпочтительнее использовать «традиционные» возможности меню.

Главное меню должно быть присоединено к окну методом *setJMenuBar*(*JMenuBar menuBar*).

Элементами главного меню являются обычные меню (выпадающие прямоугольные блоки команд – объекты класса *JMenu*). Конструктор *JMenu*(*String title*) принимает один параметр (название меню, которое будет отображаться в строке главного меню).

Меню, в свою очередь, состоит из пунктов меню, представленных классом *JMenuItem*. По логике работы пункты меню аналогичны кнопке *JButton*. При нажатии на него пользователем выполняется какое-то действие.

Создать элемент меню можно пустым конструктором *JMenuItem* либо (что более востребовано) одним из следующих конструкторов, в которые передается текст и (или) значок элемента меню: *JMenuItem*(*String text*), *JMenuItem*(*Icon icon*), *JMenuItem*(*String text, Icon icon*). В любой момент текст и значок можно сменить методами *setText*(*String text*) и *setIcon*(*Icon icon*) соответственно.

Элемент добавляется в меню методом *add*(*JMenuItem item*) класса *JMenu*. Чтобы отделить группы взаимосвязанных элементов меню, можно добавить между ними разделитель методом *addSeparator*( ) класса *JMenu*.

В меню можно добавить и другое меню. В этом случае образуется последовательность вложенных друг в друга подменю, что довольно часто встречается в современных программах. Не следует увлекаться. Глубина вложенности более трех уровней скорее всего приведет к неудобствам пользования программой.

Создадим главное меню окна, состоящее из двух подменю: *Файл* и *Правка*. В меню *Правка* поместим выпадающее подменю. Воспользу-

емся знаниями о менеджере расположения главного меню, чтобы добавить с правого края значок (наподобие того, как это сделано в браузере Internet Explorer):

```
   SimpleWindow(){
      super("Окно с меню");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      JMenuBar menuBar = new JMenuBar();
      JMenu fileMenu = new JMenu("Файл");
      fileMenu.add(new JMenuItem("Новый"));
      fileMenu.add(new   JMenuItem("Открыть",   new   ImageI-
con("1.gif")));
      fileMenu.add(new JMenuItem("Сохранить"));
      fileMenu.addSeparator();
      fileMenu.add(new JMenuItem("Выйти"));
      JMenu editMenu = new JMenu("Правка");
      editMenu.add(new JMenuItem("Копировать"));
      JMenu pasteMenu = new JMenu("Вставить");
      pasteMenu.add(new JMenuItem("Из буфера"));
      pasteMenu.add(new JMenuItem("Из файла"));
      editMenu.add(pasteMenu);
      menuBar.add(fileMenu);
      menuBar.add(editMenu);
      menuBar.add(Box.createHorizontalGlue());
      menuBar.add(new JLabel(new ImageIcon("2.gif")));
      setJMenuBar(menuBar); setSize(250,150);
   }
```

Окно с меню и подменю представлено на рисунке 35.

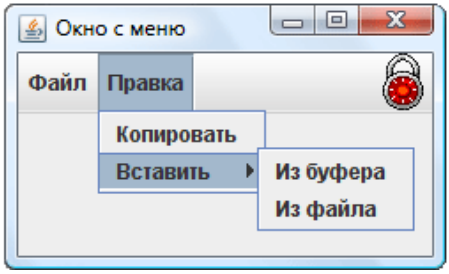

Рисунок 35 – **Окно с меню и подменю**

Например, чтобы при выборе в меню элемента *Выйти* программа прекращала свою работу, следует заменить в примере команду `fileMenu.add(new JMenuItem("Выйти"));` на следующую последовательность команд:

```
JMenuItem exit = new JMenuItem("Выйти");
exit.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event) {
      System.exit(0);
   }
});
fileMenu.add(exit);
```

Командой *System.exit*(0) прекращается работа текущего приложения.

16.40. Создание контекстного меню

Контекстное (или всплывающее) меню реализовано в классе *JPopupMenu*, который очень похож на класс *JMenu*. Отличительным методом этого класса является метод *show*(*Component comp, int x, int y*), отображающий меню в точке с заданными координатами относительно границ заданного компонента.

Контекстное меню, как правило, отображается при щелчке правой кнопкой мыши над компонентом. Чтобы отобразить меню, требуется добавить к этому компоненту слушателя мыши.

Рассмотрим пример, в котором к уже созданному окну добавляется метка с надписью и контекстное меню из двух элементов, связанное с этой меткой. Окно с меткой и контекстным меню представлено на рисунке 36.

Необходимо добавить в конструктор перед последней командой (*setSize*) следующий код:

```
   label = new JLabel("КНИЖКА", new ImageIcon("1.gif"), JLa-
bel.RIGHT);
   JPanel panel = new JPanel();
   panel.add(label);
   popup = new JPopupMenu();
   popup.add(new JMenuItem("Прочитать"));
```

```
popup.add(new JMenuItem("Сжечь"));
label.addMouseListener(new MouseAdapter(){
   public void mouseClicked(MouseEvent event) {
      if (SwingUtilities.isRightMouseButton(event))
         popup.show(label, event.getX(), event.getY());
   }
});
setContentPane(panel);
```

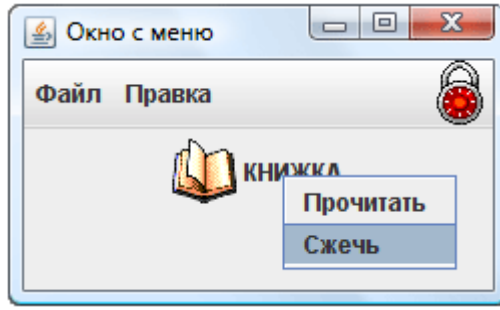

Рисунок 36 – **Окно с меткой и контекстным меню**

При этом в определении класса окна следует ввести следующие поля:

```
JPopupMenu popup;
JLabel label;
```

Это необходимо для того, чтобы анонимный класс-слушатель мог обратиться к метке и к контекстному меню.

Приведенный пример также иллюстрирует применение полезного метода *isRightMouseButton*(*MouseEvent event*), определенного в классе вспомогательных утилит *SwingUtilities*. Метод получает информацию о событии мыши и отвечает на вопрос, была ли нажата правая кнопка мыши. Методы *getX*( ) и *getY*( ) возвращают координаты курсора мыши относительно наблюдаемого компонента.

16.41. Стандартные диалоговые окна класса *JOptionPane*

Неотъемлемой частью большинства программ являются небольшие диалоговые окна (для вывода пользователю сообщения (например, сообщения об ошибке), для вопроса, ответ на который важен для выполнения текущего действия (например, просьба подтвердить или отменить запрашиваемую операцию). Эти диалоги могут быть запрограммированы вручную на основе класса *JFrame*. Ввиду того, что они являются типичными для многих программ, Swing предоставляет в распоряжение программиста несколько готовых классов для работы с ними.

Чаще всего используется класс *JOptionPane*, содержащий несколько статических методов, отображающих стандартные диалоги.

Метод *showMessageDialog*( ) выводит на экран диалоговое окно, информирующее пользователя. Оно содержит надпись, значок и кнопку *OK*. Существует несколько разновидностей этого метода с разными наборами параметров. Самый простой из них – это *showMessageDialog*(*Component component, Object content*), который требует указания компонента, над которым должно появиться диалоговое окно и содержимого окна. Чаще всего содержимым окна является некоторая строка, а вместо первого параметра передается *null*; тогда окно появляется по центру экрана. Более «продвинутый» вариант *showMessageDialog*(*Component component, Object content, String title, int type*) позволяет задать также свой заголовок окна и выбрать его тип (влияет на иконку в окне): сообщение об ошибке (ERROR_MESSAGE), предупреждение (WARNING_MESSAGE), информация (INFORMATION_MESSAGE).

Диалоговое окно является модальным. Это значит, что пока пользователь не нажмет в этом окне кнопку *OK*, программа окажется заблокирована; пользователь не сможет работать с другими окнами.

Окно сообщения системы представлено на рисунке 37.

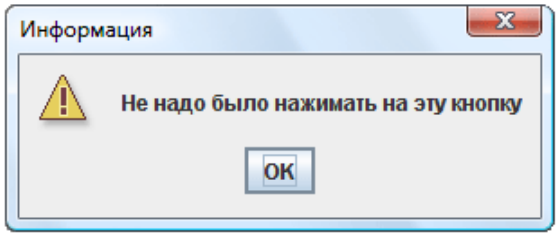

Рисунок 37 – **Окно сообщения системы**

Текст программы имеет следующий вид:

```
public class SimpleWindow extends JFrame {
   private JButton button;
   SimpleWindow(){
      super("Предупреждающий диалог");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      button = new JButton("Информация");
      button.addActionListener(
              new ActionListener() {
         public void actionPerformed(
                 ActionEvent event) {
            JOptionPane.showMessageDialog(button,
            "Не надо было нажимать на эту кнопку",
                                          "Информация",
            JOptionPane.WARNING_MESSAGE);
         }
      });
      getContentPane().setLayout(
                              new FlowLayout());
      getContentPane().add(button);
      setSize(200,150);
   }
}
```

Кнопка *button* сделана полем класса окна, чтобы можно было получить к ней доступ из анонимного класса-слушателя.

Другое часто используемое диалоговое окно – окно вопроса. В этом окне несколько кнопок, одну из которых пользователь должен нажать. В программу, вызывающую это диалоговое окно, возвращается информация о выборе пользователя, на основе которой и строится дальнейший ход работы программы.

Данное окно отображается методом *showConfirmDialog*(*Component component, Object content*). Параметры этого метода идентичны по смыслу параметрам *showMessageDialog*( ), но в диалоговом окне появится не одна кнопка, а три (*Yes*, *No* и *Cancel*). Метод возвращает значение, которое можно сравнить с константами YES_OPTION, NO_OPTION и CANCEL_OPTION. Окно вопроса представлено на рисунке 38.

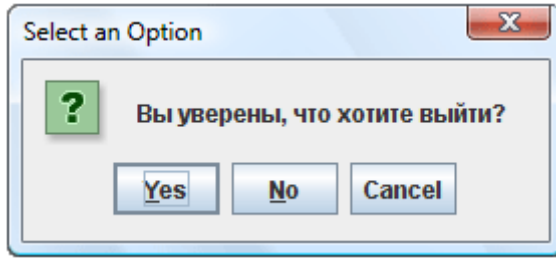

Рисунок 38 – **Окно вопроса**

Текст программы имеет следующий вид:

```
public class SimpleWindow extends JFrame {
   JButton button;
   SimpleWindow(){
      super("Предупреждающий диалог");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      button = new JButton("Выход");
      button.addActionListener(
                  new ActionListener() {
         public void actionPerformed(ActionEvent
                              event) {
            if (JOptionPane.showConfirmDialog(
               button,
               "Вы уверены, что хотите выйти?") ==
                              JOptionPane.YES_OPTION)
               System.exit(0);
         }
      });
      getContentPane().setLayout(
                           new FlowLayout());
      getContentPane().add(button);
      setSize(200,150);
   }
}
```

Метод имеет еще четыре разновидности с разным набором параметров, позволяющих изменять заголовок и значок окна, а также набор кнопок.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Блинов, И. Н. Java. Методы программирования [Электронный ресурс]. – Режим доступа : http://www.epam.by/content/dam/epam/download/book_epam_by/JAVA_Methods_Programming_v2. march2015.pdf. – Дата доступа : 02.09.2017.

Блинов, И. Н. Java. Промышленное программирование [Электронный ресурс]. – Режим доступа : http://www.epam.by/content/dam/epam/download/book_epam_by/Java. Промышленное программирование.pdf. – Дата доступа : 02.09.2017.

Блинов, И. Н. Java2. Практическое руководство [Электронный ресурс]. – Режим доступа : http://www.epam.by/content/dam/epam/download/book_epam_by/Java2. Практическое руководство.pdf. – Дата доступа : 02.09.2017.

Блох, Дж. Java. Эффективное программирование : [пер. с англ.] / Дж. Блох. – 2-е изд. – М. : Лори, 2016. – 464 с.

Васильев, А. Н. Java. Объектно-ориентированное программирование : учеб. пособие / А. Н. Васильев. – СПб. : Питер, 2013. – 400 с.

Вязовик, Н. А. Программирование на Java [Электронный ресурс]. – Режим доступа : http://old.intuit.ru/department/pl/javapl/1/. – Дата доступа : 02.09.2017.

Гудрич, М. Т. Структуры данных и алгоритмы в Java [Электронный ресурс]. – Режим доступа : http://bulletinsite.net/index.php?id1=6&category=programmer&author=gudrich-mt&book=2003&page=1. – Дата доступа : 02.09.2017.

МакГрат, М. Программирование на Java для начинающих : [пер. с англ.] / М. МакГрат. – М. : Эксмо, 2016. – 192 с.

Сьерра, К. Изучаем Java / К. Сьерра, Б. Бейтс. – 2-е изд. – М. : Эксмо, 2015. – 720 с.

Хорстманн, Кей С. Java2. Библиотека профессионала. Т. 1. Основы [Электронный ресурс]. – Режим доступа : http://cafe-aristokrat.

nethouse.ru/static/doc/0000/0000/0165/165992.a7wpu5kiws.pdf – Дата доступа : 02.09.2017.

Флэнаган, Д. Java в примерах : справ. : [пер. с англ.] / Д. Флэнаган. – 2-е изд. – СПб. : Символ-плюс, 2016. – 664 с.

Шилдт, Г. Java 8. Полное руководство : [пер. с англ.] / Г. Шилдт. – 9-е изд. – М. : Вильямс, 2015. – 1376 с.

Шилдт, Г. Java 8. Руководство для начинающих : [пер. с англ.] / Г. Шилдт. – М. : Вильямс, 2015. – 720 с.

Эккель, Э. Философия Java : [пер. с англ.] / Б. Эккель. – СПб. : Питер, 2015. – 1168 с.

СОДЕРЖАНИЕ

Учебное издание

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ высшего образования I ступени

Автор-составитель
Кравченко Светлана Витальевна

Редактор Ю. Г. Старовойтова
Компьютерная верстка Л. Ф. Барановская

Подписано в печать 30.05.18. Формат 60 × 84 $^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Ризография.
Усл. печ. л. 12,09. Уч.-изд. л. 10,10. Тираж 45 экз.
Заказ №

Издатель и полиграфическое исполнение:
учреждение образования «Белорусский торгово-экономический университет потребительской кооперации».
Свидетельство о государственной регистрации издателя, изготовителя, распространителя печатных изданий
№ 1/138 от 08.01.2014.
Просп. Октября, 50, 246029, Гомель.
http://www.i-bteu.by

БЕЛКООПСОЮЗ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ
УНИВЕРСИТЕТ ПОТРЕБИТЕЛЬСКОЙ КООПЕРАЦИИ»

Кафедра информационно-вычислительных систем

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ
высшего образования I ступени

Гомель 2018